コンピュータ・サイエンス研究書シリーズ ㊸

システム・プログラミング

David K. Hsiao 著

慶應義塾大学助教授・工博　斎藤信男 訳

日本コンピュータ協会

SYSTEMS PROGRAMMING
Concepts of Operating and Data Base Systems

DAVID K. HSIAO

The Ohio State University

原著者序文

本書の目的は，システム・プログラミングにおいてよく使われる一般的な概念を解説することである．このために，それらの概念を，抽象的に示すように努力した．抽象的に示せば，基本的な概念は，対象となる素材を詳細に論じなくても，ずっと簡単に取らえられ，理解されると思っている．時には，本書で扱っているシステムを，非常に細かく説明することもある．このような場合には，抽象的な議論と現実の世界の問題とを関連づけ，これらの概念を実際に実現することが可能であることを教えることを目指している．

ここでいうシステム・プログラミングとは，汎用のオペレーティング・システムの合成と解析とを意味している．本書では，汎用のオペレーティング・システムとして，アセンブリ・システム，入出力制御システム，一括処理システム，多重プログラミング・システムおよびデータベース管理システムだけを考えている．勿論，このほかにも，たとえば実時間システムのように，論ずる価値のあるシステムがある．しかし，それに関する素材を用意する時間や経験が無かったので，それらを本書では扱わないことにした．

システム・プログラミングの分野では，いくつか優れた理論的研究があるにも拘わらず，その一般的な理論は確立していない．したがって，ここでは，理論的に扱う努力はしなかった．そのかわり，合成および解析において，全体のシステムを論理的な構成要素に分解し，それらの要素間の関係を調べるように努めた．その過程の中で，構成要素を識別し，システムの中でのその役割を明らかにする．システムの機能を適当に仮定しておき，構成要素の論理的機能を示すことにより，それらの機能を実現させることを，時々論ずる．また，機能的に異なった構成要素を論理的に結合することによって，統一性のとれたシステムを作り出そうと試みている．分析にせよ合成にせよ，モデルや視点を選択する時には，システムの複雑さを認識するのに都合のよい枠組を与えることが

できるように努めている．入出力制御システムの層的あるいは構造的な解明，また，オフラインの一括処理システムの階層的な見方などは，このような考え方に沿ったものである．勿論，このような見解は，解析や合成のためにシステムを良く理解できる唯一の視点というわけではない．

システムの合成や解析における構造的な見方には，各層すなわち階層の各レベルを独立に扱うことができ，他の層やレベルに対する関係が，きれいな構造を持つようになるという利点がある．しかし，このほかにも，いろいろの見方がある．オペレーティング・システムの合成や解析に対して，有限オートマトンと遷移図とを用いるのも，一つの方法である．そこでは，状態，状態間の遷移，遷移をひき起す物理的な信号，遷移を促進する論理的な信号などを用いて，システムを識別し，その役割を明確にすることが，システムの解析や合成をすることになる．オペレーティング・システムを，有限オートマトンあるいは遷移図を用いて解明する方法は，まだ，流行しているわけではないけれど，非常に有力な方法になるかもしれない．ここでは，入出力制御システムの実現と，オンラインのプログラミング・システムの議論において，このような見地から論じている．また，アセンブリ・システムの記号定義の処理を理解するために，有向グラフを利用している．

知覚的な見解が使えないときは，一般的なモデルに頼っている．このようなモデルは，理想化され，ほとんどのものは，抽象的なものである．一般的なモデルを発展させることには，いくつかの利点が見られる．第一に，一見，発散して無関係な多数のアイデアが，統一のとれた提示の中で結びつけられる．第二に，実際に動いているシステムで，議論の対象となる特徴をすべて備えているものは少ない．これらのシステムの原理を支えている特徴を取り込んでおけば，そのモデルは，システムを一つのまとまりとして良く理解するために役に立つ．第三に，モデルのシステムは，実際に動いているシステムを正確に記述する必要はないので，システムを提示をするための記法や慣用にあまり制限を加えなくてもよい．このようなモデルを使って，アセンブリ・システム，多重プログラミング・システム，データベース管理システムを論じている．アセンブリ・システムは記号を処理し，データベース管理システムは，フィールド，レコード，ファイルの形をした論理的資源が主な対象であるから，抽象的なモデルを作ることに向いている．一方，多重プログラミング・システムは，物理的資源を管理するので，どちらかと言えば，抽象化には適していない．

以上の理由から，本書が目指すのは，概念や考え方を参照する枠組を与えることである．これらの見解やモデルは，システム・プログラミングを更に深く

学びたいと思う人々に対して，適切な概観と，良い洞察力を与えることができるものと願っている．何か特定のシステムの機構やその派生システムについて，深く知りたいと思っている人達を，本書の対象と考えているのではない．

第1章では，アセンブリ・システムを述べる．このアセンブリ・システムの議論は，現存のアセンブラを詳細に解明することを目指しているのではない．そのかわりに，抽象的なアセンブラを提案し，それに基づいて，アセンブリの過程における一般的な概念を定義し，そこから，アセンブラの設計を複雑なものにする要因を見つけ出す．これらの要因は，遅延型の擬似命令を処理するアセンブリ言語の機能と直接関係している．アセンブリ言語の柔軟性と能力とは，遅延型の擬似命令を処理する機能に反映されるから，アセンブラの設計における複雑さは，アセンブリ言語の柔軟性と能力とに比例すると言える．抽象的なアセンブラを調べることにより，最も能力があり柔軟であるアセンブリ言語に対して，アセンブリの過程の各段階を学ぶことが出来るだけでなく，遅延型の擬似命令を処理することに直接係わってくる段階を決定することも出来る．これらの段階を理解すれば，システムの設計者は，自分が提供したいと思っているアセンブリ言語の柔軟性と能力の程度を決めることができ，また，アセンブラの中に入り込んでくる設計および実現の複雑さの量を予想することができる．また，リンキングローダの機能についても，論じておく．アセンブリ・システムおよび入出力システムにおけるその役割を，強調してある．

第2章では，入出力制御システム(IOCS)を述べる．この章では，入出力操作の中で良く出てくる概念，すなわち，サイクルスチーリング，記憶の干渉，操作の同時性，バッファの切り替えなどについて，最初に論ずる．多重チャネルを用いた並行処理として知られている現象を提示するために，理想化したモデルを導入する．このモデルでは，システム処理量の概念と，チャネルの使用率と関係した並行性の目安とを確立する．このモデルは，少々簡略化し過ぎたきらいはあるけれども，並行処理の本質とシステムの処理量とを扱うことが可能である．実際の世界の状況と係わってくる複雑さや詳細さの程度を考えれば，簡略化し過ぎることは恐らく避けられないであろう．次に，入出力制御システムの設計をするために，2つの異なった方式を示す．すなわち，サブシステムや層を用いて入出力制御システムを設計することに中心を置いた構造的方式と，入出力制御システムを，順次型機械として構成する有限状態オートマトン方式とである．入出力制御システムは，計算機のハードウェア要素と操作とを主として扱っているから，状態および状態遷移を用いて記述しやすいので，有限状態オートマトン方式は，まだ，あまり使われてはいないけれども，入出

力制御システムの設計に対して，将来有望な方式である．入出力制御システムを，状態および状態遷移を持った抽象的な順次型機械としてとらえれば，入出力制御システムの機能とその論理とを，明確に合成し，すぐに理解し，簡単にプログラム化することができる．

本書の第3章では，一括処理オペレーティング・システムを述べる．そこでは，一般化した一括処理オペレーティング・システムを提案する．オフラインの一括処理の議論は，プロセサとサブシステムとの階層としてとらえたシステムの特徴を中心にして行なう．オンラインの入出力操作を可能にするため，また，他のシステムと結合するために，システムの能力を改善する努力についても，論ずる．オンラインの一括処理オペレーティング・システムの特徴を，状態および状態の遷移を用いて記述するのは，直ぐに可能である．また，オンライン・ジョブの入出力に対するスプーリング・システムを開発することも，重要である．

多重プログラミング・システムに関しては，2つの章を用いている．第4章では，多重プログラミング・システムの導入を促進してきた要因を識別し，また，多重プログラミングのレベルの視点から，多重プログラミング・システムを分類する．これらのレベルは，論じているシステムの工夫の仕方をかなりの程度まで，反映している．多重プログラミングの概念は容易にわかるけれども，多重プログラミング・システムの合成や解析に対する方式は，非常に多様性に富んでいることを認識しておく方がよい．これらの方式は，システムに与えられている物理的資源や，一括処理システムのような初期のシステムから設計者が得た経験などによって決められる．次に，これらの多重プログラミング・システムの実現を支えることのできるセマフォアのような機構を概括し，そのようなシステムにつきもののシステムデッドロックなどの問題点を指摘する．第5章は，仮想空間を用いた多重プログラミング・システムを中心に論じている．仮想空間の議論においては，次のような疑問に答えようとしている．すなわち，(1) なぜ，仮想空間が必要なのか，(2) 仮想空間の型には，どんなものがあり，どのような機能をもっているか，である．(1) の疑問に答えるためには，仮想空間は，記憶の動的割り付け，プログラムの再配置の繰り返し，プログラムのオーバーレイ，プログラムやデータの実行時の拡大などに関する問題を解決することを主目的として設計されていることを指摘すればよい．仮想空間を開発した人々は，上に述べた問題のうちのいくつかを解決することを目指していたので，仮想空間には，いろいろの型のものがある．このために，仮想空間の型を分類し，それらを例を用いて説明しようと思う．こうすれば，

第2の疑問に答えることができる．最後に，一般化した多重プログラミング用オペレーティング・システムを述べて，仮想記憶のオペレーティング・システムの間で良く見られる共通の特質を概括する積りである．一般化した多重プログラミング・システムの割り込み解析部とタスク開始部の役割について，概略を述べる．多重プログラミング・システムにおける物理的な入出力の操作は，(ユーザ・プログラムの代りに)常にシステムによって行なわれるので，一般化した多重プログラミング・システムにおける入出力制御システムの拡張した機能についても，論ずる．また，伝統的な論理的入出力操作と，仮想空間の環境にユニークに存在する仮想アクセス法とを比較する．動的ローディングと，多重タスキングは，また，多重プログラミング・システムにおいてユニークなものである．このようなユニークなシステムの特質を，前の章で発展させてきた素材と関連づけようと試みている．また仮想空間に付随して生ずるソフトウェアの問題点の解決法を，一般化した多重プログラミング用オペレーティング・システムを用いながら論じている．

最後の章は，データベース管理システムを扱っている．将来のオペレーティング・システムは，主としてデータベース管理を目指したものになるだろうという徴候が見られるので，これらのシステムは重要になる．計算機システムのオンライン的使用がますます強調され，オンラインの大容量記憶の費用効果比がどんどん向上しているので，集中化したデータベースと，多重アクセスの端末とを開発することが促進されてきた．したがって，従来のように，オペレーティング・システムを，計算機の物理的資源の管理者としてだけ見るのは，狭すぎる．将来の計算機システムにおけるオンライン・データベースの規模が本当に大きくなれば，オペレーティング・システムは，更に，論理的資源の管理者としても働かなければならない．論理的資源とは，主として，オンラインのデータおよびプログラムである．論理的資源の管理は，データベースにおけるデータやプログラムの構成，機密保護，アクセス法，記憶などに関するものである．データベースの管理の領域は，まだ発展の途中にあるので，本書における議論では，解決しなければならないいくつかの問題に触れることになろう．このために，一般化したデータベース管理システムに対する抽象的モデルを提案し，それを用いて，記憶のセルという概念を定義し，そこから，索引順序，多重リスト，逆ファイル，あるいは，もっと最近発表されたセル状多重リスト構造のような頻繁に使われる構造を導き出すことが出来る．

このモデルから導出できるデータ構造に対して，アクセスのアルゴリズムを与えてある．このアルゴリズムでは，その特別な性質のために，可動ヘッドの

記憶装置のアクセス機構の動きを最小化し，二次記憶へのアクセスの数を減らし，不正確なデータの検索を除去する方向を目指している．

このモデルから導出できるデータ構造に対する更新のアルゴリズムも与えてある．レコードの追加，削除，キーワードの置き換えが生ずる場合には，データ構造の動的な更新が必要になる．

また，データベース管理システムのアクセス制御と，機密の保護に対する機構も，このモデルを用いて論じている．

本書を読むためには，事前の知識として，計算機構成と，プログラミングの原理に関することが必要である．背後の素材を知るために，読者は，少くとも，C. William Gear 著の“*Computer Organization and Programming*”(McGraw-Hill 社)の最初の6章を読むか，または，Herbert Hellerman 著の“*Digital Computer System Principles*”(McGraw-Hill 社)の1, 2, 3および8章を読むと良い．また，後者の第9章には，IBM System/360 に関して，非常に簡潔であるが充分な記述がしてある．システム・プログラミングの入門の教材を更に読みたければ，John J. Donovan 著の“*Systems Programming*, McGraw-Hill 社(**訳書**　池田克夫訳 **システム・プログラム**，日本コンピュータ協会)を見ると良い．Ohio 州立大学では，Donovan 教授の教科書を，本書を用いるコースの事前のコースで，採用している．

謝　　　辞

本書は，Pennsylvania 大学の Moore School の電気工学科で，1969年秋，システム・プログラミングの最初のコースで作られてきた素材に基づいて書かれた．それは，計算機科学を専攻する大学院一年目の学生に，システム・プログラミングを教えることが目的であったが，現在，Ohio 州立大学では，学部のジュニアとシニアの学生が半分，大学院の一年生が半分いるシステム・プログラムのコースで，本書を用いている．プログラミング言語(たとえば，FORTRAN とアセンブリ言語)と，プログラミング技術に関する知識を必要とする．さらに，伝統的な汎用計数型計算機の機構と，システムのアーキテクチャとを，基本的に理解しているものと仮定している．

2, 3, 4, 5章のいくつかの部分は，合衆国科学技術財団に一部助成してもらい，Pennsylvania 大学が，1970年夏に開催した“多重プログラミング・システムの設計原理”と題した夏期プログラムの入門コースで使用したものである．このプログラムでは，五つの特定の多重プログラミング・システムを，その主設計者が，解説した．すなわち，Honeywell-GE 645 Multics を話した F. J. Corbató 教授，IBM 360/67 TSS の William L. Konigsford 氏，Univac-RCA Spectra 70/46 TSOS の Norman Weiser 氏，IBM 360/MVT の Bernard I. Witt 氏，および IBM 360 TOS/DOS の David N. Freeman 氏が，それである．そこでは，また，これらのシステムの設計を，詳細に解説する人々も招かれた．すなわち，Michael D. Schroeder 教授は，Multics について話し，Lee Varian 氏が TSS を，George Bean 氏が MVT を，John Gibson 氏が TSOS を，Gwendolyn Gartland さんが TOS/DOS について話した．彼等の講義は，特に，これらの多重プログラミング・システムを著者が理解するために大いに役立ち，本書の多重プログラミングおよび仮想記憶システムの部分全般に関して，改めることにも役立った．彼等に対し，著者は，負うところ大で

ある．

オンライン・プログラミングと，データベース・システムに関する章の教材のかなりのものは，著者の同僚と学生達の研究から得られたものであり，それは Office of Naval Research の Information Systems Program によって援助を受けた．これらの人々の熱心な研究とこの Program による援助がなければ，これらの章は書けなかったであろう．Harry A. Freedman 氏および Frank Manola 氏, Marvin Gelblat 博士, Richard P. Morton 博士, Richard L. Wexelblat 博士，Michael S. Wolfberg 博士，および，Noah S. Prywes 教授の協力に感謝したい．Office of Naval Research の Information Systems Program の Marvin Denicoff 部長には，心から感謝の意を捧げる．彼の変らぬ援助は，本書が生れることに貢献しただけではなく，オンライン・プログラミングとデータベース・システムにおけるシステム・プログラミングの現状を推進することにも寄与している．

本書は，6つの章から構成されている．各章は，いくつかの重要なシステムの概念を扱っている．そこでは，アセンブリ言語とアセンブリ・システム，入出力操作と入出力制御システム，オフラインおよびオンラインの一括処理システム，多重プログラミング，仮想記憶システム，データベース・システムを取り上げている．欠けているものとして特に注意しておくものは，コンパイラ・システムと，実時間システムである．前者を除いたのは，この件に関する教科書が多数出版されていることを考えたからである．後者を除いたのは，この主題を著者が無視したことが，主要な理由となっている．本書の素材の大部分のものは，古典的なもので，公刊された論文，レポート，本，マニュアル，ノート，私的な通信に基づいて得られたものである．情報源を確かめることおよび読者が更に研究できるようにするために，各章のおわりに“あとがき”を置き，その章での参考文献に簡潔な注釈をつけている．

また，IBM の Joel D. Aron 氏と Charles L. Gold 氏には，本書の質を非常に高める貴重なコメントと忠告を頂き，私の同僚 Douglas S. Kerr 教授には，草稿を注意深く読んで頂き，Texas 大学の Andrew Noetzel 教授には，第5章3節の初稿に貢献して頂き，Moore School の部長である John G. Brainerd 教授には，システム・プログラミングの新しいコースの必要性を認めて頂き，1965年のカリキュラムで私のコースを実現して頂いた．また，Sally Furtell さん，Cindy Karr さん，Sandy Rich さんには，草稿のいくつかの版をタイプして頂いた．合わせて，ここに感謝の意を捧げる．

前述の夏期プログラムで，これらの教材を使用したこと，および，1970年

の秋期計算機連合大会(FJCC)に招待者として講演したことが理由になり，本書の初期の草稿が，私の同僚によって，多くの大学で教科書又は参考資料として使用された．たとえば，Villanova 大学の Stephen W. Ching 教授，Massachusetts 大学の Richard Eckhous 教授，Ohio 州立大学の Clinton R. Foulk 教授，California 大学 Berkeley 分校の Lance Hoffman 教授，Delaware 大学の T. Kimura 教授，Pennsylvania 大学の Noah S. Prywes 教授，Texas 大学 A & M の Richard B. Simmons 教授らが，本書を熱心に使用して頂いたことに感謝したい．

最後に，システム・プログラミングに強い関心を持ち，本書の準備段階を価値あるものにして貰った私の学生達，および，システム・プログラミングを良く理解して，この分野で実りの多い結果と，挑戦的な問題とを与えてくれた計算機産業界の人達に対して，深い感謝の念を覚える次第である．

オハイオ州コロンブスにて
1975年7月

D. K. H.

図および表などに関する謝辞

2.4節(78ページから85ページ)における図と式とは，*Computing Survey* 誌第2巻2号(1970年7月)の H. Hellerman および H. J. Smith, Jr. による "Throughput Analysis of Some Idealized Input, Output and Compute Overlap Configurations" から借用したものである．版権は，Association for Computing Machinery 1970年であり，著者および出版元の許可を得て写している．

2.5.1節(86ページから92ページ)における例と図とは，H. Hellerman による "*Digital Computer Systems Principles*" から，少し修正して書いたものである．版権は，McGraw-Hill 出版社 1967年であり，出版元の許可を得て，使用している．

3.2.4.1および3.2.4.2節(140ページ，141ページ)の図は，R. F. Rosin による *Computing Survey* 誌第1巻1号(1969年3月)の "Supervisory and Monitor System" から採った．版権は，Association for Computing Machiney 1969年で，著者と出版元との許可を得て写している．

4.4.2節(198ページ)の眠っている医者の物語と，4.5.1.1および4.5.1.2節(200ページ)の賢明な銀行家の物語は，Academic Press 社 1968年の "*Programming Languages*" (F. Genuys 編)にある E. W. Dijkstra によって書かれた "Co-operating Sequential Processes" から応用したものである．これは，著者と出版元の許可を得ている．

4.5.2と4.5.3節(202ページと203ページ)にあるデッドロックの条件のリストと検出アルゴリズムとは，*Computing Survey* 誌第3巻2号(1971年6月)の E. G. Coffman, Jr., M. J. Elphick および A. Shoshani による "System Deadlocks" から採用したものである．版権は，Association for Computing Machinery 1971年であり，著者と出版元の許可を得て，使用している．

訳 者 序 文

本書は，システム・プログラミングの高度な教科書である．システム・プログラミングと言ってもその範囲は広いものであるが，ここでは特にオペレーティング・システムを中心として考え，その上に実現されるデータベースの基礎まで扱っている．従来のシステム・プログラミングの教科書，たとえば，J.J. Donovan の “System Programming (訳書・**システム・プログラム**，池田克夫訳，日本コンピュータ協会)” や，R. M. Graham の “Principles of Systems Programming (訳書・**システム・プログラムの原理**，有田五次郎訳，日本コンピュータ協会)” では，言語処理に関するプログラミングの扱いが多く，オペレーティング・システムに関する部分は少なかった．本書は，著者の専門を十分に活かした特色のあるシステム・プログラムの教科書であると言える．

個々の事項の記述は，抽象化，一般化をねらったモデルを想定した上で，その世界で単なる概念の説明だけにとどまらず詳細な解析やアルゴリズムの展開を目指している．多少まわりくどい点も見られるが，抽象的モデルの上でこれだけ詳細に論じられるのは，オペレーティング・システムやデータベースに関する著者の造詣の深さを示すものであろう．

著者 David K. Hsiao 教授は，現在，オハイオ州立大学におられ，オペレーティング・システムおよびデータベースの分野で活躍されている．1970 年の FJCC で，ペンシルバニア大学の J. W. Carr 教授と共にオペレーティング・システムに関する招待講演をされたことがあり，たまたま訳者はそれを聴く機会があった．最近は，むしろデータベース関係に力を入れているようである．ACM 発行のデータベースに関する専門の雑誌 TODS (Transactions on Database System) の初代の Editor-in-Chief になっており，また，データベースの VLDB (Very Large Data Base) 国際会議やソフトウェア工学の国際会議の中心的役割も果している．

1975 年の第 2 回日米コンピュータ会議のために Hsiao 教授が来日した際，本書の翻訳の件などに関して話す機会があった．そこで，同教授に「何故データベースの基礎までこの教科書で扱っているのか」と訊ねたところ，「将来の

オペレーティング・システムはデータベース向きに設計され実現されることになるからだ.」と明快に答えられた. この考え方は, 本書の最後に第7章むすびとして書かれている. 将来, 情報システム/計算機システムのユーザは, データベース管理システムのインターフェースを介してシステムを使うようになることは確かであろう. したがって, 現在のユーザとオペレーティング・システムとのインターフェースがデータベースとのインターフェースに移行すると思えば, 基本システム・ソフトウェアは当然データベースを実現し易いように設計されなければならない. このような考えを背景にして, 第6章データベース管理システムを読んで貰えば, 著者の意図が正しく理解できるであろう.

最近, 計算機関係の術語もかなり日本語化されてきており, 訳語の選択においてもそれらを考慮の対象にした. しかし, まだ原語をすぐに連想するのが難しそうなものも多かったので, 片かなで表わした術語が相当多くなってしまった. 重要な術語に関しては, 最初に現われたときに原語を示してある. 本来, 日本語として読んで計算機に関する概念が理解できるのであれば, 原語が連想できなくても構わないはずであるが, 翻訳という立場も考慮してやや中途半端な結果となった次第である. 術語の問題に関しては, 読者のご叱正とご批判とをお願いしたい.

本書の翻訳を著者にまで約束しておきながら, 個人的理由で大幅に遅れてしまったことを誠に申し分けなく思っている. その間, 終始激励とご協力を頂いた日本コンピュータ協会の村越勝弘氏に感謝の意を捧げる次第である.

1980年1月

斎 藤 信 男

目　次

原著者序文 ……iii
謝　辞 ……ix
図および表などに関する謝辞……xii
訳者序文 ……xiii

第1章　アセンブリ・システム ……*1*

注　意 ……*1*
1.1　はじめに ……*1*
1.2　アセンブリ・システム ……*2*
1.3　アセンブリ言語 ……*3*
1.3.1　基本形式 ……*3*
1.3.2　記号機械命令 ……*5*
1.3.3　擬似命令 ……*9*
1.3.3.1　ロケーションカウンタとロケーションの割り当て ……*9*
1.3.3.2　コントロールセクションと記号ディクショナリ ……*11*
1.3.3.3　データの定義と生成 ……*13*
1.3.3.4　プログラムとリスティングの制御 ……*14*
1.3.3.5　記号の定義 ……*14*
1.3.4　遅延型記号定義 ……*15*
1.4　アセンブリ ……*17*
1.4.1　字句解析：入力プログラムの文字列の走査とそのコード化の過程 ……*17*
1.4.2　構文解析：正しいプログラムの命令を決定する過程 ……*19*
1.4.3　意味処理：記号の定義と擬似命令の評価の過程 ……*21*
1.4.3.1　ディクショナリ ……*21*
1.4.3.2　有向グラフによる見方 ……*28*
1.4.3.3　3段階の処理 ……*31*
1.4.3.4　第2パス：定義の代入と式の評価 ……*33*
1.4.4　目的プログラム生成の過程 ……*38*
1.4.5　結　論 ……*39*
1.5　リンキングローダ ……*42*

1.5.1　再配置と記憶の利用 …… 43
1.5.1.1　チェイニングとオーバーレイ …… 43
1.5.1.2　動的ローディング …… 44
1.6　あとがき …… 46
参考文献 …… **46**
演習問題 …… **48**

第2章　入出力操作と入出力制御システム …… **55**

注　意 …… 55
2.1　入出力操作の例 …… 59
2.2　バッファの大きさとバッファの割り当て …… 61
2.2.1　釣り合いのとれたプログラムを完成できるか …… 61
2.2.2　バッファの割り当てアルゴリズム …… 67
2.3　入出力干渉と入出力機構の利用 …… 70
2.3.1　記憶サイクルスティーリングの現象 …… 70
2.3.2　CPU を入出力操作の一部として使うこと …… 71
2.3.3　データチャネル …… 72
2.3.4　釣り合いのとれたチャネルと機器の利用 …… 74
2.4　入力，出力および計算操作の重ね合わせにおけるバッファの割り当てとチャネル多重度の影響 …… 75
2.4.1　2つのモデルと4つの場合 …… 76
2.4.1.1　事例Ⅰ：順次処理 …… 78
2.4.1.2　事例Ⅱ：単一チャネルを使った並行処理 …… 78
2.4.1.3　事例Ⅲ：2つのチャネルを使った並行処理 …… 80
2.4.1.4　事例Ⅳ：2つのチャネルを使った高度な並行処理 …… 81
2.4.2　処理量の公式 …… 81
2.4.2.1　計算限界のプログラムに対する効果 …… 84
2.4.2.2　入出力限界のプログラムに対する効果 …… 84
2.5　入出力制御システム …… 86
2.5.1　有限状態オートマトンとしての IOCS の設計 …… 86
2.5.2　処理系とサブシステムの層としての IOCS の設計 …… 92
2.5.2.1　入出力操作，チャネルおよび機器に関する情報の構成 …… 93
2.5.2.2　IOCS の4つの基本機能 …… 96
2.5.2.3　物理的および論理的入出力制御システム …… 100
2.6　あとがき …… 103
参考文献 …… **103**
演習問題 …… **104**

第3章　一括処理オペレーティング・システム ……………………………… ***107***

注　意……………………………………………………………………………………107
3.1　はじめに……………………………………………………………………………108
3.2　処理系および副処理系の階層：オフライン一括処理オペレーティング・システムの見方……………………………………………………………………………111
3.2.1　プログラミング言語処理系のオフライン一括処理オペレーティング・システム……………………………………………………………………………114
3.2.1.1　階層の根……………………………………………………………………114
3.2.1.2　階層のほかの部分：非常駐型制御プログラム……………………………118
3.2.1.3　コマンド言語………………………………………………………………119
3.2.2　オフライン・オペレーティング・システムの構成要素……………………120
3.2.2.1　スーパーバイザ……………………………………………………………120
3.2.2.2　モニタ………………………………………………………………………122
3.2.2.3　サブシステム………………………………………………………………122
3.2.3　オフライン一括処理オペレーティング・システムの要約……………………131
3.2.3.1　例……………………………………………………………………………132
3.2.3.2　ジョブ制御言語の利用……………………………………………………136
3.2.4　オフライン一括処理オペレーティング・システムにおけるいくつかの改良……………………………………………………………………………………139
3.2.4.1　オフラインジョブ入力……………………………………………………140
3.2.4.2　直接結合システム…………………………………………………………140
3.3　オンライン一括処理システム……………………………………………………142
3.3.1　簡単なオンライン一括処理システム…………………………………………143
3.3.1.1　表，バッファおよびコマンド……………………………………………143
3.3.1.2　衛星計算機のオペレーティング・システム：少数の状態をもった有限状態オートマトン…………………………………………………………………146
3.3.1.3　中央計算機のオペレーティング・システム………………………………147
3.3.1.4　ファイルシステム…………………………………………………………152
3.3.2　オンライン一括処理オペレーティング・システムの要約といくつかの改良……………………………………………………………………………………152
3.3.2.1　間接結合システム…………………………………………………………153
3.3.2.2　遠隔ジョブ投入……………………………………………………………154
3.3.2.3　システムの構成……………………………………………………………155
3.4　あとがき……………………………………………………………………………166
参考文献 ……………………………………………………………………………… ***166***
演習問題 ……………………………………………………………………………… ***168***

第4章　多重プログラミング ……………………………… ***172***

4.1　注　意 ……………………………… *172*
4.2　なぜ多重プログラミングか ……………………………… *172*
4.3　ロード時再配置可能多重プログラミングシステム ……………………………… *177*
4.3.1　ジョブだけの多重プログラミング ……………………………… *177*
4.3.1.1　入出力制御ブロックと IOCS との構成 ……………………………… *178*
4.3.1.2　ファイルの保全性と入出力デッドロックに関する考察 ……………………………… *181*
4.3.1.3　オペレーティング・システムの階層構造 ……………………………… *182*
4.3.1.4　要　約 ……………………………… *185*
4.3.2　ジョブおよびジョブ内のプログラムに対する多重プログラミング ……………………………… *185*
4.3.2.1　タスクおよびサブタスクから見たオペレーティング・システムの構成 ……………………………… *187*
4.3.2.2　実行中のタスクの同期 ……………………………… *188*
4.3.2.3　共通資源へのアクセスの制御 ……………………………… *190*
4.3.2.4　資源割り当て戦略 ……………………………… *191*
4.3.2.5　制限された競争 ……………………………… *195*
4.3.3　結　論 ……………………………… *196*
4.4　同期と相互ロックの機構 ……………………………… *197*
4.4.1　セマフォと同期基本命令 ……………………………… *197*
4.4.2　眠っている医者：一つの例 ……………………………… *198*
4.4.3　要　約 ……………………………… *199*
4.5　システムデッドロック ……………………………… *199*
4.5.1　システムデッドロックの問題 ……………………………… *199*
4.5.1.1　銀行の問題 ……………………………… *200*
4.5.1.2　銀行の問題の解 ……………………………… *200*
4.5.2　システムデッドロックの特徴づけとその防止 ……………………………… *202*
4.5.3　システムデッドロックの検出と回復 ……………………………… *203*
4.5.4　結　論 ……………………………… *205*
4.6　多重プログラミング環境におけるプログラムの構成 ……………………………… *205*
4.6.1　再入可能型プログラムの結合の約束 ……………………………… *207*
4.6.2　再入可能型サブルーチンの結合の約束の例 ……………………………… *209*
4.7　あとがき ……………………………… *210*
参考文献 ……………………………… ***211***
演習問題 ……………………………… ***212***

第5章　仮想記憶オペレーティング・システム …… ***218***

注　意 …… *218*

5.1　仮想空間と番地変換の概念 …… *219*

5.1.1　動的記憶割り当て …… *219*

5.1.1.1　記憶利用の要求 …… *219*

5.1.1.2　プログラムの断片化と詰めの操作 …… *219*

5.1.1.3　動的記憶割り当てとプログラム再配置の繰り返し …… *221*

5.1.1.4　仮想番地，仮想空間および番地変換 …… *223*

5.1.1.5　変換関数を構成する際の実用的考慮 …… *224*

5.1.2　プログラム・オーバーレイのない仮想空間を目指して …… *227*

5.1.2.1　線形仮想記憶システム …… *228*

5.1.2.2　線形セグメント化仮想記憶システム …… *232*

5.1.3　まばらに使用される線形セグメント化仮想空間 …… *235*

5.1.3.1　セグメンテーションと単層記憶の概念 …… *235*

5.1.3.2　まばらに使用される線形セグメント化仮想記憶システム …… *237*

5.2　一般化した多重プログラミング用オペレーティング・システム …… *239*

5.2.1　ハードウェアへの要求 …… *240*

5.2.1.1　処理装置 …… *240*

5.2.1.2　物理的記憶 …… *241*

5.2.1.3　補助記憶 …… *242*

5.2.1.4　入出力チャネルと機器 …… *242*

5.2.2　ソフトウェアシステムの構成 …… *243*

5.2.2.1　オペレーティング・システムの常駐部 …… *245*

5.2.2.2　物理入出力操作と PIOCS …… *250*

5.2.2.3　論理入出力操作，LIOCS とファイルシステム …… *258*

5.2.2.4　仮想アクセス法 …… *269*

5.2.2.5　仮想空間の節約 …… *273*

5.2.2.6　動的結合 …… *275*

5.2.2.7　多重タスキング …… *284*

5.2.3　要　約 …… *286*

5.3　あとがき …… *287*

参考文献 …… ***288***

第6章　データベース管理システム …… ***290***

注　意 …… *290*

6.1　はじめに …… *290*

6.2　データ構造……293
6.2.1　相似性……293
6.2.2　可変構造……294
6.2.2.1　レコードの構成……295
6.2.2.2　ファイル構造……311
6.3　問い合わせ……327
6.4　ディレクトリのデコーディングとファイルの探索……329
6.5　2つのアクセスアルゴリズム……331
6.5.1　順次アクセスアルゴリズム……332
6.5.2　並列アクセスアルゴリズム……335
6.5.3　3つの例……343
6.6　更　新……349
6.6.1　3つの更新アルゴリズム……349
6.6.2　構造的な移動の現象……352
6.6.3　屑集め系……352
6.6.3.1　屑集め系の役割り……352
6.6.3.2　屑集めのアルゴリズム……353
6.6.4　番地の単調性……357
6.7　データの機密保護：論理的アクセス制御の研究……358
6.7.1　宝石商の問題……358
6.7.2　アクセス制御と保護の3つのレベル……360
6.7.3　論理的アクセス制御モデル……362
6.7.3.1　データベースの論理的要素の識別……362
6.7.3.2　アクセス型の表現……364
6.7.3.3　所有権の概念……365
6.7.3.4　データ管理コマンドとマクロの使用の基礎……366
6.7.3.5　機密保護の手続き……370
6.7.4　データの機密保護に関する要約……376
6.8　あとがき……377
参考文献……378
演習問題……380
第7章　むすび……383
索　引……387

第 1 章

アセンブリ・システム

注　　意

アセンブリ・システムに関する本章を読む人は，たとえば，[3][†]や[25]に書かれている簡単な2-パスアセンブラの基礎は，良く知っているものと仮定している．

この章の内容が進むに従って，読者は，更に学習するために，残りの参考文献を見たくなるかもしれない．ここに含まれている内容と，参考文献との関係についての注釈は，章の最後に述べてある．

1.1 はじめに

アセンブラは，[3]や[25]で述べているように，アセンブリ言語の記号機械命令を処理するために主として設計されたものであるが，その複雑さは，多くの場合，アセンブリ言語の擬似命令における柔軟さに直接比例して決まる．記号機械命令とは異なり，擬似命令には，対応する機械命令がない．これは，プログラマがアセンブラと連絡をし，記号機械命令を用いたプログラミングが上手に出来るような手段として，提供されている．擬似命令が柔軟性に富み，それが非常に使いやすいならば，アセンブラの設計や実現が，非常に複雑なものになる．

アセンブラを，2パスの処理系と見る伝統的な考え方は，簡略化し過ぎている．擬似命令と入力したプログラムの前処理とについて論じた後で，伝統的な2パスにはあまり似ていない抽象アセンブラの内部的な働きを示して，この事実をあきらかにしよう．抽象化をすることによって，読者は，アセンブラの機能の本質を，どのようにコード化するかということにとらわれずに，集中的に

---

† かっこ内の番号は，章の末尾にある参考文献の通し番号を示している．

理解することができる.

なお，この章の残りの節では，アセンブリ・システムにおけるリンキングローダの役割について述べる.

1.2　アセンブリ・システム

アセンブリ・システムは，一つのプログラミング・システムであるから，その入力として，アセンブリ言語で書かれた**原始プログラム**を受け付け，アセンブリ言語の構文と意味とに基づいてプログラムを翻訳し，その出力として，**目的プログラム**を作り出す．アセンブリ言語は，計算機の操作に正確に対応する記号命令と，データの生成と記憶の割り付けを直接行なう擬似命令とから構成されているので，アセンブラは，データの生成，記憶の割り付け，計算機の操作の使用などの点で，コンパイラの支援システムとして使用することもできる．このために，あるシステムのアセンブラでは，ユーザでもコンパイラから

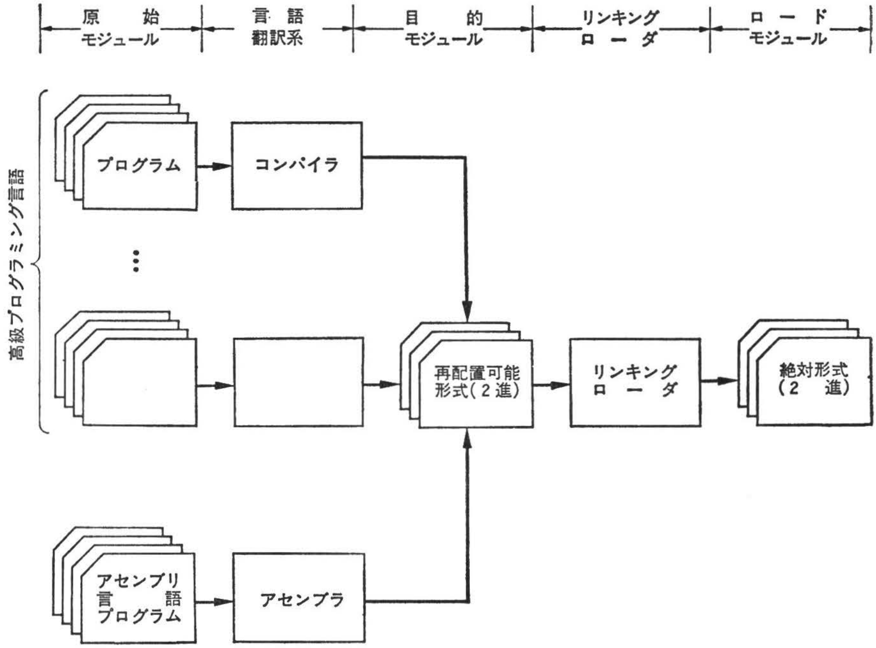


図 1.1(a)　アセンブリ・システムの例(1)

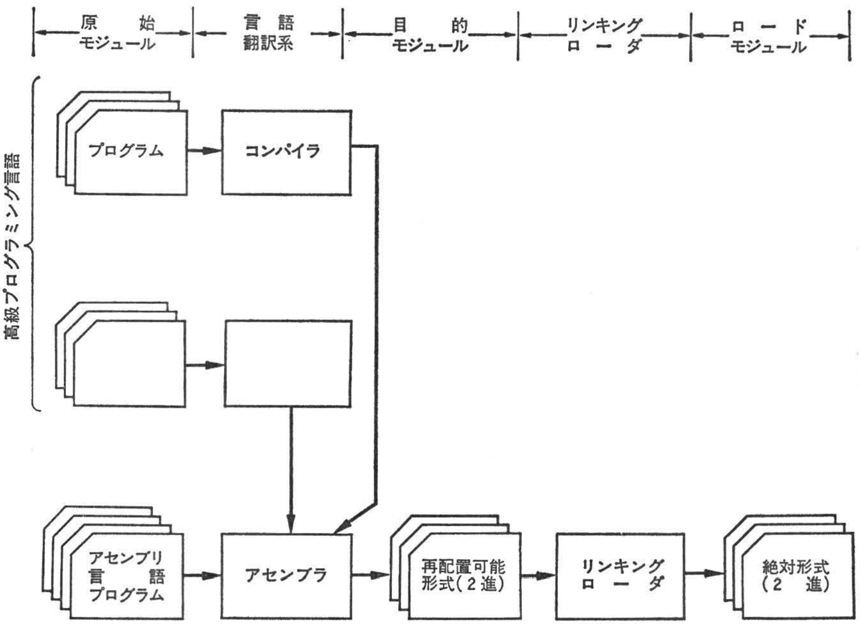


図 1.1(b) アセンブリ・システムの例(2)

でも原始プログラムを受け入れることができる．**図 1.1 (a)** および**図 1.1 (b)** には，2つのアセンブリ・システムの構成が示してある．たとえば，IBM 360/370 のアセンブリ・システムは，前者に対応するのに対し，IBM 7090/7094 は，後者に対応する．

1.3 アセンブリ言語

1.3.1 基本形式

アセンブリ言語の命令の基本形式は，次のとうりである．

ラベル	オペレーション	オペランド

ラベルは，選択的につけられる記号名である．オペレーションも記号であり，必ずなければならない．特に，機械命令に対応するものがある記号のオペレーションを，**記号機械命令** (mnemonic machine instruction) と呼び，対応する

機械命令のない記号オペレーションを，**擬似命令**(pseudoinstruction)と呼ぶ．オペランドが必要かどうかは，オペレーションの構文の規則による．オペランドが必要であろうとなかろうと，それは，記号名であってよい．たとえば，次の例は，IBM 360/370 のプログラムセクションである．

ラベル	オペレーション	オペランド	コメント
NAME	CSECT		NAMEは，このプログラム・セグメントを指す
BEGIN	BALR	TWO,Ø	レジスタ TWO を用いた Branch and link
	USING	*,TWO	レジスタ TWO の内容は，ロケーション・カウンタの現在値である．
	⋮		
TWO	EQU	2	TWO は，値2をとる．
	END	BEGIN	このプログラム・セグメントの終りを示し，この最初の実行可能な命令は BEGIN にあることを示す．

命令 CSECT, USING, EQU, および, END は，擬似命令であり，BALR は，記号機械命令である．ここには，3つの記号ラベル，すなわち，NAME, BEGIN および TWO がある．さらに，IBM 360/370 アセンブリ言語 [9] の構文では，CSECT 命令のオペランドと，END のラベルは，空でなければならない．

次に，IBM 7090/7040 のプログラムセグメントを示す．

ラベル	オペレーション	オペランド	コメント
Z	BES	N	N語の記憶域を予約する．(N+1)番目の語を，Zでラベルを付ける．
	⋮		
	USE		主ロケーション・カウンタを，以下の命令のために用いる．
X	DEC	1,2	2つの10進数1と2とを定義する．
	CLA	X	アキュムレータを介して，Clear and add をする．
W	STO	Z − 1	アキュームレータの内容を，Z-1 に格納する．
	⋮		
N	EQU	W − X	Nの値は，(W-X)の値に等しい．
	END		

ここで，BES, USE, DEC, EQU および END は，擬似命令であり，CLA と

STO は，記号機械命令である．Z, X, W および N は，記号ラベルである．アセンブリ言語については，読者は，文献 [8] を参照されるとよい

1.3.2 記号機械命令

アセンブリ言語の記号機械命令は，アセンブラの設計の複雑さに，重要な影響を与えることはない．実際，アセンブリ言語が，記号機械命令だけから成り立っているのならば，その言語のアセンブラの設計は，簡単な練習問題になる．この場合，アセンブラは，単に，次のような仕事をするだけでよい．

1. ロケーションカウンタの用意をし，初期値を与える．
2. アセンブリプログラムを，一時に，一記号命令ずつ読み込む．
3. プログラムを，凝縮した形式で格納する．
4. 命令の記号の正当性を調べる．
5. 命令に対して，使われているオペランドの形を区別する（すなわち，記号名，絶対番地，相対番地，リテラルなど）．
6. 表の中に，記号名を入れる．
7. その表の中に，命令のラベルとして使われている記号名を入れ，その名前に，ロケーションカウンタの現在値を割り当てる．
8. ロケーションカウンタの値を，命令の長さだけ増加する．
9. アセンブリ言語プログラムの終端を，見つける．
10. すべての記号命令を，対応するコードに置き換える．
11. すべての記号名を，表の中で割り当てられた値にする．
12. 正しくない記号命令や，値の割り当てられない記号名があれば，誤りとその診断メッセージを与える．
13. 相対または絶対番地であるという情報，および，リテラルとして指示されている値に関する情報を用いながら，機械命令のプログラムを出力する．

1. では，ロケーションカウンタが，ハードウェアのプログラムカウンタのソフトウェア版であることに注目しよう．1.，2.，3.，4.，6. より，アセンブラは，ロケーションカウンタの作成，原始プログラムと凝縮したプログラムに対するファイル，正しい記号命令のリスト，記号名および記号ラベルに対する表のために，作業領域が必要になることがわかる．そのような表を，一般に，**記号表**(symbol table) とよぶ．

記号命令の正しさの検査は，大抵，アセンブリの最初の段階で行なわれる．

さらに，正しい記号命令のリストは，通常，記号表の中に入っている．これらの記号は，記号表に埋め込まれたものと考えた方が良い．正しくない記号コードが検出されたならば，そのアセンブリの処理は，すぐに，放棄できる．一方，6. と 7. の仕事の処理の仕方には，かなり柔軟性がある．あるアセンブラでは，すべての記号ラベルを集めて，それらに対するロケーションカウンタの値の割り当てが済むまでは，オペランド内の記号名を集めない．別のアセンブラでは，記号名と記号ラベルを同時に集めて，可能な限り，それらに値を割り当てようとする．オペランドの中の記号名は，ラベルとして現われ，7. でロケーションカウンタの値を割り当てられることに注目しよう．したがって，記号名は，ラベルと同じ値をとる．このような場合，記号名は，解決した(rosolved)という．解決していない記号名は，プログラム内のどこにもラベルとして現われなかったので，割り当てられた値のない記号である．

通常は，図1.2(a)および図1.2(b)に示してあるように，1. から 9. の仕事は，2パスアセンブラの第1パスに，また，10. から 13. の仕事は，第2パスに相当する．

要約すると，アセンブリの処理では，次のような基本的機能を実行する．

パスI

a）初期設定――記号表およびロケーションカウンタの設定，および，原始プログラムと凝縮したプログラムのファイルのオープン

b）検査――命令コードの正当性のチェック

c）識別――オペランドの型および型の組み合せの決定

d）割り当て――新しい記号の処理およびその記号に対するロケーションカウンタの現在値の割り当て

e）後始末――パスIの仕事の終了と，パスIIの処理の始動

パスII

f）置き換え――記号命令と，対応する(2進)コードとの置き換え

g）解決――表の記号の値の決定と，記号と定数との組み合せに対する値の計算

h）メッセージ――診断メッセージとリスティングの生成

i）出力――原始プログラムの(2進)表示の生成

2パスアセンブラの設計と実現とは，簡単で素直に行なえるけれども，このアセンブラに対するアセンブリ言語は，非常に制限の強いものとなる．これら

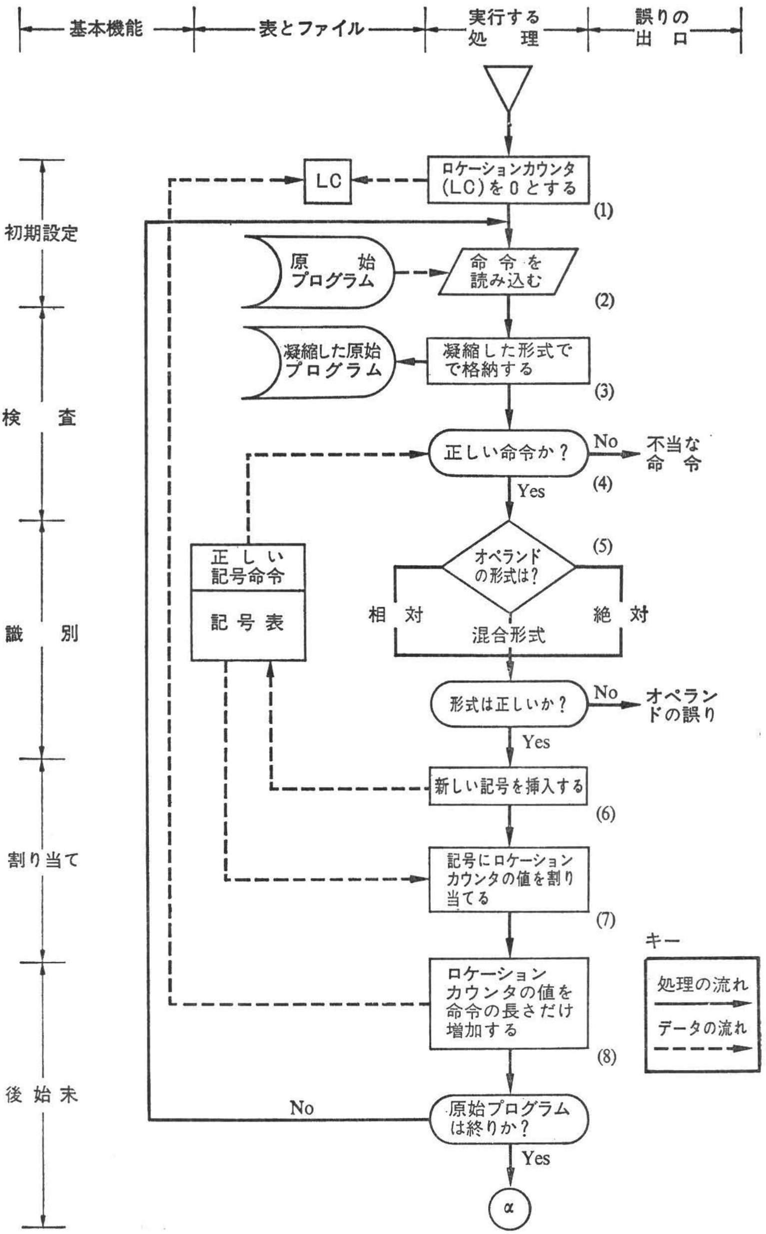


図 1.2(a) アセンブリのパスⅠ

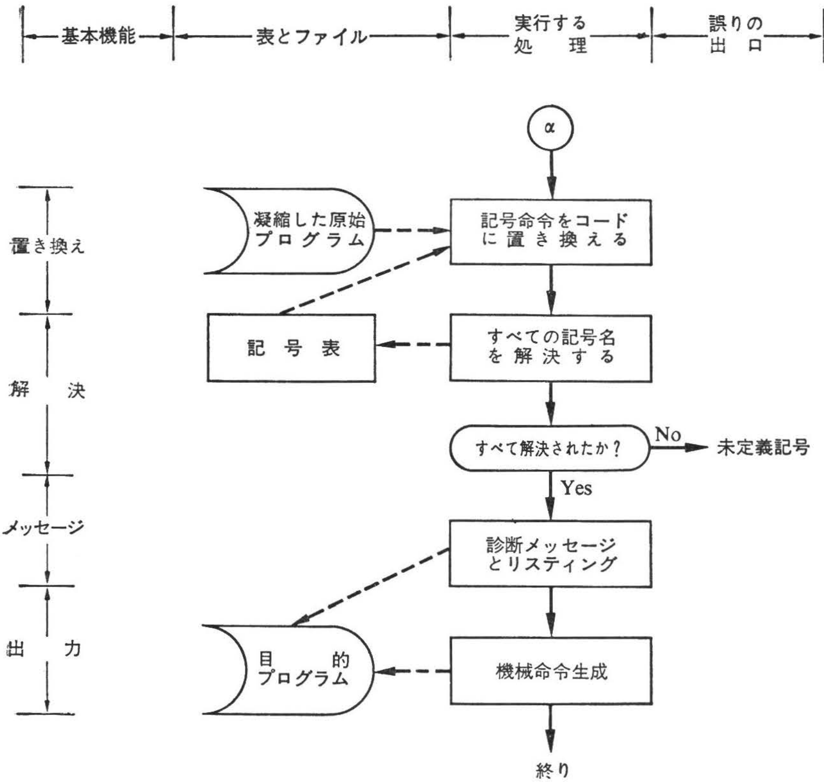


図 1.2(b)　アセンブリのパスII

の制限のために，ユーザは，次のことができなくなる．

1. プログラムの命令を一つの順番に従って書き，同じプログラムを，別の順番に従って実行すること．
2. 独立に記述したプログラムを，一つに集めて，一緒に実行すること．
3. リテラルや定数を，EBCDIC や ASCII でコード化せずに実現すること．
4. 作業領域を確保すること．
5. プログラムのリスティングに関して，アセンブラに伝えること．

1.3.3 擬似命令

擬似命令を用いれば，プログラマは，自分のプログラムの記号機械命令のアセンブリに関し，必要なことをアセンブラに伝えることができる．したがって，擬似命令により，プログラマは，アセンブリの過程を制御することができる．たとえば，どのアセンブリ言語にも，End-of-Program という擬似命令があるが，これによって，プログラムの中には，命令がもはや存在しないということをアセンブラに伝える．擬似命令の種類は，システムによってそれぞれ異なっている．しかし，ここでは，擬似命令を，その機能の点から述べてみよう．説明のためには，例として，IBM 360/370 と IBM 7090/7040 のアセンブリ言語を引用する．

1.3.3.1 ロケーションカウンタとロケーションの割り当て

前節でわかったように，現在処理されている命令のロケーションが，命令群の先頭からどの位になっているのかを保持するために，ロケーションカウンタを用いる．複数個のロケーションカウンタが使用できる場合には，(原始モジュール内の) いくつかの命令群を，あらかじめ決めておいた実行順序になるようにアセンブルさせることを，プログラマはアセンブラに指示できる．

通常，いろいろのロケーションカウンタが制御している命令の再配置は，ロード時にリンキングローダが行なっている (ロードすることおよびリンキングローダについては，1.5 節で述べる)．すなわち，ロケーションカウンタによって，ユーザは，一つの順序に沿って書いた命令を，リンキングローダを介して別の順序にロードすることができる．

そこで，一つのロケーションカウンタの制御のもとにある命令群は，ローディングの一単位になる．いくつかのアセンブリ・システムでは，このような命令群は，**コントロールセクション**(control section)を成すという．すなわち，このようなアセンブリ・システムでは，各コントロールセクションにロケーションカウンタが割り当てられ，そのコントロールセクション内での命令のロケーションを管理している．しかし，プログラマは，そのコントロールセクションに割り当てられたもの以外には，新しいロケーションカウンタを用いることは許されていない．別のアセンブリ・システムでは，一つのコントロールセクションは，複数個の命令群から構成されていてもよく，それぞれの群は，独自のロケーションカウンタによって管理されている．これらの方法は異なっているけれども，それらは同じ効果をもつ．プログラム内のいくつかのコード群の実行の順序を指定するために，前者の場合はユーザは各単位をコントロールセ

クションとして定義すればよく，後者の場合は，各単位に対し，一つのロケーションカウンタを割り当てればよい．ロケーションカウンタを陰に割り当てるか，陽に宣言するかどちらにしても，全体的な効果としては，コード群の順序を指定している．

コントロールセクションに割り当てられたロケーションカウンタ（すなわち主ロケーションカウンタ）以外の特別なものを使うためには，プログラマは，擬似命令でロケーションカウンタの指定をしなければならない．

ロケーションカウンタとロケーションの割り当てとを操作するために，以下に示すような機能をもつ擬似命令が必要になる：

1. 命令の順序を制御するためのロケーションカウンタに名前をつける．
2. ロケーションカウンタに，初期値を割り当てる．
3. ロケーションカウンタの現在の値を変更する．

たとえば，IBM 7090/7040 アセンブリ言語では，擬似命令

ラベル	オペレーション	オペランド
	USE	LC1

は，LC1 という名前のついた新しいロケーションカウンタを作るか，または，既に作られている LC1 という名前のロケーションカウンタに切り換えることを示す．この USE 以降にあるすべての命令は，LC1 の制御下におかれる．擬似命令

ラベル	オペレーション	オペランド
	BEGIN	LC1, 5

は，ロケーションカウンタ LC1 に，初期値 5 を割り当てる．この BEGIN は，ロケーションカウンタの現在値を変更するときにも使える．

IBM 360/370 アセンブリ言語では，ロケーションカウンタの作成は，陰に行なわれる．コントロールセクションの開始と共に，そのコントロールセクションに対して，一つのロケーションカウンタが作られる．たとえば，命令

ラベル	オペレーション	オペランド
LC1	CSECT	

は，LC1 という名前のついたコントロールセクション(したがってロケーションカウンタ)が作られる．命令

ラベル	オペレーション	オペランド
LC1	START	5

は，LC1 に初期値5を割り当てるのに使われる．START は，一番先のコントロールセクションでのみ使用できる．ロケーションカウンタの値を変更するためには，ORG が使われる．たとえば，命令

ラベル	オペレーション	オペランド
	ORG	* − 5

は，現在のロケーションカウンタの値を，5だけ減らす．BEGIN はプログラムのどこでも使えるが，IBM 360/370 アセンブリ言語のORG は，値を変えるべきロケーションカウンタに対応するコントロールセクション内で使わなければならない．

1.3.3.2 コントロールセクションと記号ディクショナリ

大きなプログラムを書くときには，プログラマは，そのコードを分割して，各部分を別々にアセンブルされるコントロールセクションとすることができる．これらのコントロールセクションは，どのような順序で実行してもよく，それらが終了すればデバッグができる．そして，それらのコントロールセクションをすべてデバッグしテストを完了すれば，全体のプログラムをロードして実行することができる．

一つのプログラムをコントロールセクションに分割すれば，多くの利点がある：

1. 個々のコントロールセクションのコーディングとデバッギングとが並行的に実施でき，問題の解析から実行するプログラムが使えるまでの時間が短縮される．
2. プログラムの変更は，通常少数のコントロールセクションに限定されるので，プログラム全体を再アセンブルすることなく変更が可能となる．
3. 以前にチェックが完了しているコントロールセクションの利用，すな

わちサブルーチンのライブラリの利用により，労力の重複が最小に押さえられる．

4. 大きなプログラムのいくつかのコントロールセクションをコーディングするために異なったプログラミング言語が使用できるので，これらの言語の長所をプログラマが利用できる．

コントロールセクションを操作するために，

1. コントロールセクションの開始を指定すること，
2. コントロールセクションの続きを指定すること，

ができるような擬似命令が必要になろう．前節に示した擬似命令の例は，コントロールセクションの操作を説明している．

プログラム間の通信には，一つのプログラムから別のものへ制御が移ることや，共通のデータを参照することがある．アセンブラは，このような通信の手段を提供しなければならない．さらに，大きなプログラムは，すべて同時にアセンブルされるわけではないが，いくつかのコントロールセクションから構成されているかもしれない．したがって，アセンブラは，独立にアセンブルされたコントロールセクションの間の通信の方法や手段をさらに提供しなければならない．そのようなプログラムは，一つまたはそれ以上のコントロールセクションから構成されている．独立にアセンブルされたコントロールセクション間で行なわれる通信の機構は，プログラムをコントロールセクションと見なすことができるので，プログラム間の通信の場面にも適用できる．

この通信を実現するために，プログラマは，擬似命令を使って，次のことをアセンブラに対して行なう：

1. 一つのコントロールセクションの中で，他のコントロールセクションから参照されても構わない記号ラベルを識別すること．
2. (独立にアセンブルされた)他のコントロールセクションの中で，このコントロールセクションから参照をする記号ラベルを指定すること．

そこで，アセンブラは，上記の情報を含む**外部記号ディクショナリ**(external symbol dictionary)をアセンブルされたコントロールセクションに対して作り出す．

外部記号ディクショナリを用いれば，(1.5節で述べる)リンキングローダは，ロードする前に，相互参照を解決することができるであろう．

たとえば，IBM 360/370 のアセンブリ言語では，擬似命令

ラベル	オペレーション	オペランド
	ENTRY	LC2, LC3

は，このコントロールセクション内の記号ラベル LC2 と LC3 とが，他のプログラムやコントロールセクションから参照するオペランドとして使用できることを指示する．擬似命令

ラベル	オペレーション	オペランド
	EXTRN	LC3, LC4

は，2つの記号ラベルが，ENTRY 名またはコントロールセクション名であることを示す．勿論，これらのラベルは，どこか別のコントロールセクションで定義されている．

IBM 7090/7040 における ENTRY および EXTRN の使い方も，上と同様である．

1.3.3.3 データの定義と生成

記憶内にデータを導入し，プログラム内に記憶領域を確保するために，次のような種類の擬似命令が通常用意されている：

1. 定数の定義：　いろいろの型と長さの定数．
2. 記憶領域の定義：　いろいろの長さの記憶．

IBM 360/370 と IBM 7090/7040 のアセンブリ言語は，どちらも強力なデータの定義と生成の命令をもっている．たとえば，IBM 360/370 では，定数定義と記憶領域定義の擬似命令は，次に示すように，いろいろのデータ型を指定することができる．

ラベル	オペレーション	オペランド	コメント
	DC	C'AB',	文字AとBとを生成する．
		X'FE',	16進数FEを生成する．
		B'1Ø1',	2進数101を生成する．
		F'12',	10進数12を2進一語に変換する．

		E'46.2',	10進数46.2を，仮数部24ビット，指数部7ビットの浮動小数点一語として生成する．
		A(LC1),	LC1の番地の値を生成する．
		V(SINE)	外部記号の番地がおかれる．
AREA	DS	80C	80文字の記憶領域を予約する．先頭の文字の記号番地は，AREAである．

IBM 7090/7094アセンブリ言語に対しては，3つの例を選んで説明しよう．

ラベル	オペレーション	オペランド	コメント
S	DEC	1,2,3,4,	10進数1,2,3,4を2進数に変換し，連続した場所に格納する．記号Sは，これらの場所の先頭を示す．
AREA	BSS	5	現在のロケーションカウンタに5を加えて，5つの場所を確保する．記号AREAは，これらの場所の先頭を示す．
END	BES	6	記号ENDは，6つの場所を確保した後の最初の場所を示す．

1.3.3.4 プログラムとリスティングの制御

ある擬似命令は，空白の設定，見出しや副見出しの印字，頁送りなど，プログラムのリスティングを制御するために用いる．プログラムの終了や，リテラル・プールの開始の指定など，原始プログラムに対する制御命令もある．

1.3.3.5 記号の定義

データ定義の擬似命令のように，記号定義の擬似命令により，プログラム内に記号を導入し定義することができる．通常，命令のラベルとして記号を書いただけで，その記号は定義される．勿論，この記号の定義は，命令を制御して

いるロケーションカウンタの現在値となる．この場合には，記号定義のための擬似命令を使用する必要はない．

一つの記号に対して，ロケーションカウンタの正当な値以外のものを割り当てたいときには，記号定義のための擬似命令を必ず使わなければならない．

IBM 360/370 および IBM 7090/7040 のアセンブリ言語では，EQU 擬似命令を使う．たとえば，

ラベル	オペレーション	オペランド
A	EQU	B

では，記号 A に対し，既に B に割り当てられているものと同じ値が割り当てられる．

1.3.4 遅延型記号定義

次の例に示すような EQU 擬似命令の使用を，もう一度考えてみよう．

ラベル	オペレーション	オペランド
LENGTH	EQU	LISTTOP − LISTEND + 1

LENGTH は，先頭が LISTTOP で終端が LISTEND であるようなリストの長さとして定義しようとしていることはあきらかである．しかし，LISTTOP と LISTEND が事前に定義されていなければ，IBM 360/370 のアセンブリ言語では，このような命令は不当なものになる．

その理由としては，LENGTH の値は LISTTOP と LISTEND の値に基づいて決まるのであるから，この命令の処理の前にこれらの値がわからなければ，その差をアセンブラは計算できないからである．これらの記号が両方とも，プログラムの後の部分でちゃんと定義されることがわかっていても，何の助けにもならない．言いかえれば，このアセンブラは，遅れて定義されるような記号の処理はできない．このような記号を，**遅延型の記号**(symbols of deferred type)と呼び，オペランドに遅延型記号をもつ擬似命令を，**遅延型の擬似命令**(pseudo-instructions of deferred type)と呼ぶ．記号を遅延型にしない一つの方法は，(この例にある)EQU 命令を，プログラムのちょうど終りにもってくることである．しかし，これは，プログラマにとって不便であるばかりでなく，たとえば，プログラマが，パラメータとしてリストの長さを必要とするような汎用のリス

ト追跡プログラムを書いているときには，その正しい長さ(すなわち，その値)は，わかるはずがない．

同じ理由により，IBM 360/370 アセンブリ言語の擬似命令 ORG は，次のように使うことはできない．

ラベル	オペレーション	オペランド
	ORG	LENGTH

ここで，LENGTH は，事前に定義されていないものとする．

IBM 360/370 のアセンブリ言語で遅延型の記号の定義の処理に柔軟さが欠けるのは，主として，IBM 360/370 のアセンブラが“2パス”アセンブラとして構成されているからである．

2パスアセンブラでは，次のような系列の命令を処理することはできない．

ラベル	オペレーション	オペランド
	ORG	A
A	EQU	B
B	EQU	5

記号 A は，未だ定義されていないので，1パス目では擬似命令 ORG は処理できない．同様にして，EQU 命令のラベル A は，記号 B のために定義できない．しかし，B は，このパスで定義されるかもしれない．2パス目では，A が未だ定義されていないので，ORG の処理はやはり引き延ばされる．それにもかかわらず，このパスで前の EQU 命令があらわれたときには，B が既に定義されているので，ラベル A は定義できる．A をオペランドとしてもつ ORG 命令が処理できるのは，3パス目である．このような考察から，遅延型の記号を扱えるのは，多重パスのアセンブラであることがわかる．遅延型の記号を扱えるアセンブラがあれば，プログラマは，新しい記号を，必ずしも事前に定義していない記号を用いて導入することができる．遅延型の記号を自由に導入できるならば，アセンブリ言語は非常に柔軟性に富んだものとなる．しかし，また，アセンブラがずっと複雑な(多くのパスがある)ものになる．次節では，遅延型の記号の処理ができる抽象的な多重パスアセンブラについて述べよう．これは多少複雑ではあるけれども，このアセンブラは一般的なモデルとして使用でき，そこから特別な場合として2パスアセンブラを抽出できる．

1.4 アセンブリ

多重パスアセンブラに対する基本的なアセンブリの過程は，次の5つの機能から構成される．このうち，4つの過程については，次節で詳細に述べる．

1. **字句解析** アセンブリの過程におけるこのフェーズの主な機能は，入力プログラムの文字列の走査とそのコード化である．コード化されたものは，通常非常に圧縮された形になり，アセンブリの過程の他のフェーズの処理系で簡単に操作できる．
2. **構文解析** アセンブリの過程におけるこのフェーズでは，コード化された個々の文を調べて，その文がアセンブリ命令として正しいか正しくないかを決定する．文のオペランドの中には，数値，リテラルや記号名から成る算術式やその他の式から構成されているものがあるので，構文解析が必要となる．
3. **意味処理** アセンブリの過程の中で，とび抜けて複雑なフェーズは，コード化された正しいアセンブリ命令に対して意味処理を行なうことである．ここでは，2つの作業が必要である．第一に，記号の定義の作業である．一つの記号を定義するとは，決められた方式に従って，その記号に値を割り当てることである．その値が，他の記号の値から導かねばならないとき，割り当ての作業はかなり複雑なものになる．このような割り当ては，擬似命令によって，ほとんど決まる．それゆえ，擬似命令の評価が，意味処理の主要な作業となる。
4. **プログラムの生成** 入力プログラムに対する目的プログラムの生成が，この過程の仕事である．
5. **プログラムのリスティングとエラー・メッセージ**

1.4.1 字句解析： 入力プログラムの文字列の走査とそのコード化の過程

アセンブラに対する入力は，プログラマが書いたもの(図1.1a参照)か，または，FORTRANやCOBOLのコンパイラのような言語処理系が生成したアセンブリ言語のプログラム(図1.1b参照)である．どちらの場合にも，アセンブラに対する入力を，入力原始プログラムあるいは略して**入力プログラム**とよぶ．コンパイラの生成した入力プログラムの場合には，コンパイラは前処理系として働くので，字句として正しくコード化された入力プログラムをアセンブラに与えることができる．したがって，アセンブラは，コンパイラの生成した入力プログラムに対して字句解析をする必要はない．一方，ユーザが与える入

力プログラムに対しては，字句解析は充分行なわなければならない．入力プログラムは，記号の定義や擬似命令の評価に対してあまり関係のない情報を大量にもっているので，アセンブラがそれを繰り返し使用することは滅多にない．たとえば，文の中の注釈やたくさんの空白は，プログラマにとっては有益であるけれども，プログラムのリスティング以外では，アセンブラに対して役に立つことはない．実際，文字をスキップしたり検出したりするのには時間がかかるので，アセンブリの過程にそれらは邪魔になる．

アセンブリの処理を行なうために，入力プログラムを一回だけ読む．読み込みが終ると，コード化した形式の入力プログラムを生成し，アセンブリの以後の過程で何回も使用する．

ここで，入力プログラムのコード化の方法を述べよう．

1. 入力プログラムの各文を読み込み，その文中の記号を型別に識別する．
2. 各記号に対し，**記号表**（図 1.3 参照）を探索する．記号表のエントリ

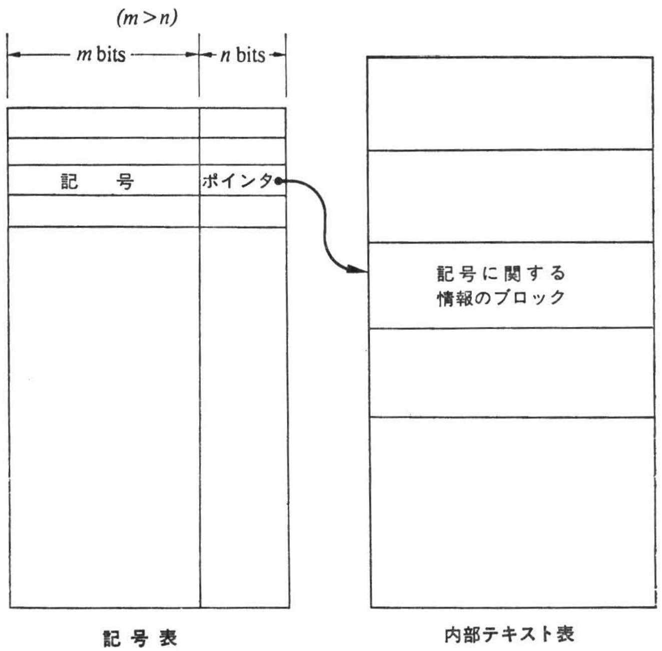


図 1.3　記号表と内部テキスト表

は，2つの要素，すなわち記号とポインタとから構成されている．ポインタは，その記号を記述する情報のブロックを指している．記号に関する情報は**内部テキスト表**(internal text table)と呼ばれる表に格納してある．探索で見つからなければ，記号表に新しい記号を登録する．新しい記号 S_i を記号表に入れるときには，その記号に関する情報を生成し，内部テキスト表に入れる．その情報に対するポインタ P_i を作り，記号表内のその記号につけて格納する．そうでなければ，記号 S_i は，記号表の中に存在する．そのポインタ P_i を用いて，内部テキスト表内のその記号に関する情報のブロックを更新する．この更新のときに，多重定義の記号などのようなある種の条件を，検出することができる．

3. つぎに，文の中の記号 S_i を，それに対応するポインタ P_i で置き換える．文は，(a) その文中のすべての記号が対応するポインタに置き換えられた，(b) すべての空白列が，空白の指示子と空白のカウンタとで置き換えられた，(c) 元のテキスト中に注釈があるときには，元の入力プログラムの注釈欄に対するポインタが生成された，ときに**コード化された**という．

このようにして，入力プログラム内のすべての記号は，対応するポインタと置き換えられる．その利点は，次の通りである．

1. コード化した入力プログラムは，元のものに比べて，短かい．
2. S_i のかわりに P_i を用いれば，記号の定義の処理において，アセンブラは，内部テキスト表内の S_i に関する情報を P_i を介して直接読み出すことができる．

2番目の利点は，読者にとって，明白でないかもしれない．これを，もう少し詳しく述べよう．コード化した入力プログラムの中で，ポインタ P_i のかわりに記号 S_i を使っているとしたならば，アセンブラが，たとえば記号 S_5 に出会うごとに，記号表の中でそれを探索しなければならない．この表の中から，記号 S_5 に関する情報へのポインタ P_5 をアセンブラは求めることができる．しかし，S_5 は，いろいろの文の中で記号名として何回も使用してもよいので，S_5 に対する探索を繰り返すのは，時間がかかってしまう．

1.4.2 構文解析：正しいプログラムの命令を決定する過程

アセンブリ命令の構文は単純であるので，アセンブリの過程における構文解析は，主として，オペランド内の算術式の解析を行なうことになる．ここでは，

解析アルゴリズムや構文解析系を学ぶつもりはない．これらについては，文献にたくさん発表されている．ここで注意すべきことは，アセンブリ言語が異なれば，構文の規則が大幅に異なるという事実である．たとえば，IBM 360/370のアセンブリ言語では，**正当な**(valid)式に対する構文規則は，次のとおりである．

1. 単純式は，記号(先頭が英字である1から8個の英字又は数字)，きちんと大きさのきまる値(たとえば，B'010' とか C'ABC')，ロケーションカウンタの参照(たとえば，ロケーションカウンタの現在値を示す * の使用)，または，リテラル(たとえば，10進35に対する固定小数点1語の2進表示を意味する ＝F'35')のどれかだけである．
2. 式は，単純式か，または，式あるいはかっこのついた式を算術的に結合したものである．
3. 結合のための算術演算子は，**挿入型演算子**(infix operator)だけに限る．すなわち，加法(＋)，減法(−)，乗法(*)，および，除法(/)である．
4. 算術演算子は，2つ続けて使用することはできない．
5. 式は，16個以上の単純式から構成することはできない．
6. 2個以上の単純式から構成される式は，リテラルを含んでいてはならない．
7. 一つの式は，5レベル以上のかっこを使うことはできない．

これによれば，次に示す式は，**不当な**(invalid)ものである．

−A＋BCD	負の符号が挿入型演算子ではないので，規則3.に反する．
A**B	規則4.に反する．
A＝F'0123'	2つの単純式Aと＝F'0123'とが，算術演算子で結合されていないので，規則1.，2.および6.に反する．
A＋(B＋(C−D*(H/(F＋(2−G)))))	規則7.に反する．

アセンブリの過程における構文解析では，命令内の式が正当か不当かを判断できるけれども，その式の値が存在するかどうかを予見することはできない．記号の定義，リテラルの定義やロケーションカウンタの値を用いて式の評価をするためには，アセンブリは，意味処理のフェーズに入らなければならない．記号やリテラルの定義，および，ロケーションカウンタの値が決まり，その定義や値の算術的演算が評価されるのは，そのフェーズである．次節において，意味処理を論ずるときに，式の評価の問題に戻ることにしよう．

1.4.3 意味処理：記号の定義と擬似命令の評価の過程

ここでは，記号の定義および擬似命令の評価の過程の大部分を扱かう．それらに関して，少し詳しく述べてみよう．便宜上，正当にコード化した入力プログラムを，単に，**プログラム**と呼ぶことにしよう．さらに，コード化したプログラムを読者が読むのは難しいので，ここでは，コード化した形式は用いないことにしよう．そのかわりに，意味処理を論ずるために，元の形式を用いる．したがって，ここで記号 S_i といったときは，アセンブラは，実際には，それに対応するポインタ P_i を使用していると考える．

最初に，内部で用いる表と，一般的な過程について述べよう．この過程は複雑なので，読者は，忍耐強く読んでほしい．しかし，グラフ的な見方と詳細な例とを用いて，この過程を説明しよう．

1.4.3.1 ディクショナリ

プログラム内の記号の処理に際しては，アセンブラは，記号表およびその内部テキスト表のほかに，記号に関する情報を蓄えておく表をたくさん作る．一つの記号は，まだ定義されていない他の記号を用いて定義してもよいから，1.3.4 節で示したように，記号がすべて定義されるまでに，何回も記号を調べる必要がある．このために，入力プログラムのかわりに小形で簡潔にコード化した内部表を用いるならば，記号の探索と処理において時間の節約が大幅に可能となる．

基本的には，記号の定義の過程において，3つの表がさらに使用される．それらは，次のものである．

1. 命令ディクショナリ(略して *I*-ディクショナリ),
2. ロケーションカウンタ・ディクショナリ(*L*-ディクショナリ),
3. 擬似命令ディクショナリ(*P*-ディクショナリ).

これらのディクショナリは，論理的には別物であるけれども，一つの実現の中では，物理的に結合されたものであってよい．一般に，これらのディクショナリのエントリは，(コード化した)入力プログラムを最初に読み込んだときに作られた組(tuples)の形をしている．これらの組に名前をつけるために，*I*-ディクショナリの i 番目のエントリを I_i, *L*-ディクショナリの j 番目のディクショナリを L_j, *P*-ディクショナリの k 番目のエントリを P_k と表わす．

命令ディクショナリ

概念的には，この過程では，プログラム内のラベルのついた各命令に対して一つのエントリを作る．すなわち，プログラム内で参照される各命令が，このディクショナリの中へ書かれる．ラベルのつかない命令は，この過程では飛ばされる．しかし，飛ばした命令の長さと，これらの命令を制御しているロケーションカウンタとは，このエントリの中に示しておく．もっと詳しく述べれば，プログラムの i 番目のラベルつき命令に対して，

$$I_i(d,\ p)$$

というエントリをもつ．ここで，d は，i 番目および $(i-1)$ 番目のラベルつき命令の間の（命令の長さの和として定義される）距離であり，p は，同一のロケーションカウンタで制御されているものの次のエントリへのポインタである．次のエントリが無い場合には，p は L-ディクショナリのロケーションカウンタのエントリを指している．

たとえば，次に示す IBM 360/370 のアセンブリ・プログラムに対して，$I_0(b,\ I_1)$ というエントリが作られる．これは，最初のラベルつき命令を示し，その前の（ラベルのない）命令の長さが 6（すなわち 6 バイト）であり，同じロケーションカウンタで制御されているラベルつき命令の次のものは，同じディクショナリの 2 番目のエントリであることを示している．2 番目のエントリ I_1 は，名札が BADDR である命令に対して作られる．

	ロケーション	ラベル	オペレーション	オペランド	コメント
			CSECT		
	000000		STM	14,12,12(13)	レジスタの内容を退避.
	000004		BALR	10,0	ベースのロード.
			USING	*,10,9	
			USING	STORAGE,12,11	
I_0 →	000006	TT	L	9,BADDR	
	00000A		B	TT1	
I_1 →	00000E	BADDR	DC	F'12'	
			⋮		

別の例として，次に示す IBM 7090/7040 アセンブリプログラム・セグメントに対して，$I_0(0,\ I_1)$ と $I_1(,\ L_0)$ なるエントリを持つディクショナリを考えよう．2 番目のエントリは，2 番目のラベル付き命令であることを示し，これと

最初のラベル付き命令との間の距離がこの時点ではわからないこと，また，このラベル付き命令は，最初(0番目)のロケーションカウンタによって制御されるものの最後の命令であることを示す．

	ロケーション	ラベル	オペレーション	オペランド	コメント
			USE		空白(主)ロケーションカウンタを使う．
I_0 →	000000	ORIGIN	CLA	X	アキュムレータを払い，Xの内容をそれに加える．
	000001		ADD	Y	Yの内容をアキュムレータに加える．
	000002		STO	Z	和をZに格納する．
	000003		TRA	W	Wへ飛ぶ．
I_1 →	?	Z	BES	N	
			USE	M	ロケーションカウンタMを使う．
			CLA	X	
			⋮		

命令の長さが決まらないのは，記号Nがあとから定義されるからである．Nの値がわからなければ，確保すべき場所の数がわからない．Zは，確保したブロックのすぐ次の場所をあらわすので，Nが定義されるまでZの値を決めることができない．したがって，ラベルZにより，*I*-ディクショナリの中に命令に対するエントリが作り出されるけれども，その前のラベルORIGINとの間の命令あるいは場所の長さを推定することはできない．I_1におけるポインタのエントリがL_0であるのは，それ以後の命令の制御のために，別のロケーションカウンタが使われるからである．USE Mという擬似命令は，IBM 360/370アセンブリ言語におけるM CSECT命令に非常に似ており，残りの命令の制御は，Mという別のロケーションカウンタで行なうことを示している．

ラベルつき命令が，機械命令であろうと擬似命令であろうと，*I*-ディクショナリの中に一つエントリが作られることに注意しよう．エントリのポインタ(すなわち第二の成分)により，同一のロケーションカウンタの下にあるすべてのラベル付き命令が結合される．そして，その結合の最後のポインタは，その結合リスト内の命令を制御しているロケーションカウンタ(*L*-ディクショナリ内のエントリ)を指している．

擬似命令ディクショナリ

P-ディクショナリは，擬似命令に対するエントリを持っている．これらのエントリは，

$$P_k(d,\ p,\ `t')$$

の形式をしている．ここで，d は前述のような定義であり，p は，そのエントリが同一のロケーションカウンタの下にあるときは，I-ディクショナリ内のエントリへのポインタ，擬似命令によりロケーションカウンタの初期設定が行なわれるときは，L-ディクショナリ内のロケーションカウンタへのポインタである．また，t は，擬似命令のテキストである．

前の節でもみてきたように，擬似命令は，主としてロケーションカウンタあるいはデータや記号の生成に関して，アセンブラに与える指令(directive)である．前者の場合，ポインタは，関係するロケーションカウンタをとらえていなければならない．このために，p は，L_j，すなわち，L-ディクショナリの j 番目のロケーションカウンタのエントリを指している．後者の場合，アセンブラは，データや記号が生成される場所を知っていなければならない．その擬似命令の直前にあるラベル付き命令を見れば，データや記号の生成が始まる場所を決定できる．したがって，この場合，p は，I_i，すなわち，I-ディクショナリの最後のエントリを指している．さらに，d は，ラベル付き命令と擬似命令との間の距離を示していることを知っている．擬似命令に対するエントリが作られた時に d の値を決定できない場合は，この擬似命令は遅延型である．このことについて，あとで例を用いて説明しよう．

擬似命令にラベルがついているときには，特別な配慮が必要である．完全を期するために，ラベルに対して，I-ディクショナリにエントリを一つもうける．そして，擬似命令に対する P-ディクショナリのエントリ中のポインタ p は，I-ディクショナリ内のそのエントリを指す．ここで，その距離 d は勿論零である．ラベルつき擬似命令が遅延型であるならば，I-ディクショナリのエントリ中の距離 d(P-ディクショナリのエントリ中の距離 d と混同せぬよう)は，のちほど決めてもよい．

すなわち，$P_k(d,\ p,\ `t')$は，

$$P_k(d,\ L_j,\ `t')$$

かまたは

$$P_k(d,\ I_i,\ `t')$$

のどちらかの形式をしている，

たとえば，Z の値をロケーションカウンタ M に割り当てるような次の擬似命令を考えてみよう．

ラベル	オペレーション	オペランド
	BEGIN	M, Z

P_1(, L_2, 'BEGIN M, Z')

というエントリが，P-ディクショナリ内に生成されるであうう．このエントリは，距離は未定(実際には，必要がない)であること，ロケーションカウンタ M か，ポインタ L_2 で指されるようなエントリを L-ディクショナリにもっていること，および，命令のテキストを以後の評価のために保存してあることを示している．P_1 の添字は，後で評価が必要なプログラム内の 2 番目の擬似命令であることをあらわしている．

もう一つの命令

ラベル	オペレーション	オペランド
	BSS	5

に対して，

P_2(2, I_0, 'BSS 5')

というエントリが生成されるであろう．この場合，ラベルのない擬似命令は，ロケーションカウンタと無関係であることを，このエントリが示している．さらに，先行するラベルつき命令との距離は 2 である．これは，ポインタ I_0 によって参照するから，そのラベルつき命令は，プログラム内で最初(0 番目)のラベルつき命令であることがわかる．

また，次のようなラベルつき擬似命令を考えてみよう．

ラベル	オペレーション	オペランド
Z	BES	N

これに対して，

$$P_3(0,\ I_1,\ \text{'BES N'})$$

というエントリが生成されるであろう．この例では，この擬似命令はやはりロケーションカウンタと無関係であることを示している．この擬似命令はラベルがついているので，I-ディクショナリ内に，ラベルに対するエントリが必要になる．そのエントリは，I_1 によって指されていることがわかる．I_1 と P_3 のエントリは，同一の擬似命令に対してとられているので，距離が零であることは別に不思議ではない．しかしながら，

$$I_1(d,\ I_2)$$

のエントリの中の距離 d は，数 N が未だ決っていないならば，知ることができない．擬似命令が遅延型の場合，このようなことが生ずる．

t の中には，ロケーションカウンタ(L_i と指定されるもの)や，命令(I_i と指定されるもの)を処理する方式に関して，アセンブラに対する指令が書かれていることを，もう一度考えよう．この指令は，アセンブラによって解釈しなければならない．アセンブラの意味処理における解釈は，それをすべてソフトウェアで行なうのが普通であるから，遅くて複雑なものになる．このため，いわゆる‘速い’アセンブラでは，擬似命令の評価を最小に保つために，遅延型の擬似命令を，出来る限り少なくしている．あきらかに，この方法を用いると，アセンブリ言語の豊かさや柔軟さが大幅に制限されてしまう．

プログラミング言語の命令のアセンブリや翻訳をもっと高速にするために，多重ロケーションカウンタとか，データや記号の生成など，ソフトウェアで用いる概念に対応するハードウェアを提供するように設計されている商用計算機システムは少ない．そのかわりに，プログラム(ロード)モジュールの実行の高速化を目指している．しかし，これらの機能として，計算機システムは，ユーザがアクセスできるプログラムカウンタや，データを記述できるいくつかの機械命令が実現されているだけでよい．APL 機械のような高級言語向き機械は，この種の機能をいくつか持っているけれども，それらはまだ実験的なものである．このような機能を持った機械について読みたい読者は，文献 [10] を参照するとよい．

ロケーションカウンタ・ディクショナリ

最後に，L-ディクショナリの作成を述べよう．L-ディクショナリは，ロケーションカウンタに対するエントリから構成されているので，L-ディクショナリのエントリと，入力プログラム内で使用するロケーションカウンタとは，

一対一の対応がつけられる．各ロケーションカウンタに対して，4つの情報項目が必要である．

1. ロケーションカウンタの初期値．
2. ロケーションカウンタが制御している命令．
3. 直前のロケーションカウンタ．
4. 直後のロケーションカウンタ．

ロケーションカウンタの初期値設定は，擬似命令により行なわれる場合，先行するロケーションカウンタの最後の値により行なわれる場合，および，主ロケーションカウンタ[†]であるときは，アセンブラによって行なわれる場合があることに注意しよう．

これらのことを考慮に入れると，*L*-ディクショナリのエントリは，次のような形式になる．

$$L_j(m,\ p',\ p'',\ p''')$$

ここに，ロケーションカウンタが *P*-ディクショナリの擬似命令エントリによって初期設定されるときは $m=1$，それ以外の場合は $m=0$ である．

p' は，そのロケーションカウンタが制御している *I*-ディクショナリまたは *P*-ディクショナリ内のエントリへのポインタである．

p'' は，次のロケーションカウンタを初期設定する *P*-ディクショナリの擬似命令エントリへのポインタである．

p''' は，このカウンタの最終値によって初期値が設定されるようなロケーションカウンタへのポインタである．

上の定義から，次のことがわかる．

1. 同一のロケーションカウンタの下にある(命令に対する)エントリは，それらが，*I*-ディクショナリ，*P*-ディクショナリ，*L*-ディクショナリのどこに分散していようとも，そのポインタによって**リング**(ring)をなすようにつながっている．
2. *L*-ディクショナリ内のロケーションカウンタのエントリは，主ロケーションカウンタのエントリが**リスト** (list) の先頭にくるようにつながっ

† ハードウェアのプログラムカウンタ(PC)を一つだけ持つような順次型機械においては，主ロケーションカウンタは一つしかない．他のすべてのロケーションカウンタはこれに続くものである．多数の PC を使える並列型機械では，主ロケーションカウンタは複数個あってよく，それぞれが他のいくつかのロケーションカウンタを先行してもよい．しかし，主ロケーションカウンタは，並列処理や制御をシミュレートしているのであるから，お互に先行したり従属したりすることはない．

ている．

さらに，リストの中の各エントリが，リングの源となっていることがわかる．したがって，エントリ中のポインタ(すなわち，I_i と P_k エントリの p, L_j エントリの p', p'', p''')が，ロケーションカウンタのエントリのリストを構成し，かつ，個々のロケーションカウンタの制御している命令のリングを構成しているのである．次節で，このリングとリストについて説明しよう．

後節で述べる反復過程を用いると，プログラム内のすべての記号を定義するために，リストの最初のエントリから始めてそのリング中のすべてのエントリをたどり，さらに，リストの次のエントリとそれに付随したリングをたどり，というような操作ができる．

1.4.3.2　有向グラフによる見方

最初に，これらのディクショナリと，反復的定義アルゴリズムの流れを説明するために，次のような例を考えよう．

ラベル	オペレーション	オペランド	コメント
	USE	H	ロケーションカウンタHを使う．
	TRA	Y	
Z	BES	N	N語の記憶ブロックを確保する．(N+1)語目をZとラベルづける．
	BEGIN	M,Z	Zの値をロケーションカウンタMへ割り当てる．
	USE		主(空白)ロケーションカウンタを使う．
X	DEC	1,2	2個の10進数1と2とを定義する．
	CLA	X	
W	STO	Z − 1	
	USE	M	ロケーションカウンタMを使う．
Y	NOP	0	
N	EQU	W − X	Nの値は(W−X)の値と等しい．
	END		

ここには，記号番地が Z, X, W, Y, N である5個のラベル付き命令と，名前が H, M および空白であるような3個のロケーションカウンタとがあることに注意しよう．さらに，3個の擬似命令がある．第1の擬似命令

ラベル	オペレーション	オペランド
Z	BES	N

は，Nが未だ定義されていないので，遅延型である．したがって，ラベルZは定義できない．第2のもの

ラベル	オペレーション	オペランド
	BEGIN	M,Z

は，Zが未定なので，やはり遅延型である．第3のもの

ラベル	オペレーション	オペランド
N	EQU	W − X

は，現在のロケーションカウンタの値を用いないで，(W − X)の値をNに割り当てることを示す．この例題プログラムに対するディクショナリは，次のようになる．

P-ディクショナリ	*L*-ディクショナリ	*I*-ディクショナリ
P_0 (0, L_0, 'BEGIN, Ø') P_1 (, I_0, 'BES N') P_2 (, L_2, 'BEGIN M, Z') P_3 (0, I_4, 'EQU W − X')	L_0 (1, I_1, , L_1) (空白ロケーションカウンタに対して) L_1 (0, P_1, P_2,) Hに対し L_2 (1, I_3, , 0) Mに対し	I_0 (8, L_1) Zに対し I_1 (0, I_2) Xに対し I_2 (3, L_0) Wに対し I_3 (8, L_2) Yに対し I_4 (3, 0) Nに対し

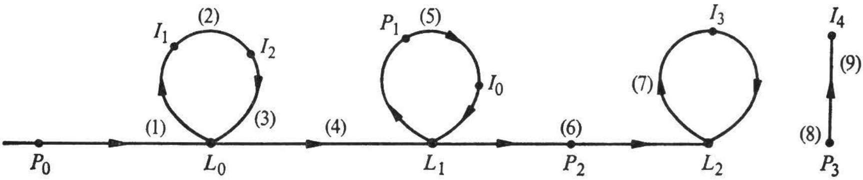
上述のディクショナリは，エントリの最終的な内容(図1.7参照)をあらわしている．5個のラベル付き命令があるので，*I*-ディクショナリには5個のエントリがある．さらに，*L*-ディクショナリの3個のエントリは，入力プログラム内で宣言された3個のロケーションカウンタに対応する．入力プログラム内の3個の擬似命令に対応する3個の*P*-ディクショナリのエントリは，p_1，p_2およびp_3である．この図から，次のことがわかる．

1. 入力プログラムの命令に対する何らかのエントリを作る前に，この処

理は，L-ディクショナリに L_0 エントリを入れる．これは，ユーザが何かロケーションカウンタを宣言しようとしまいと，この処理が，常に主(空白) ロケーションカウンタをプログラムのために作ることを意味する．

2. 主ロケーションカウンタの初期設定を行なうために，P-ディクショナリ P_0 のエントリに示すように，ロケーションカウンタを初期設定するめの(入力プログラムにはない)擬似命令を作り出す．

エントリ中のポインタは，記号の定義および擬似命令の評価のための反復定義アルゴリズムによって使用される．定義アルゴリズムは，有向グラフの追跡者，すなわち，矢印(ポインタ p, p', p'', p''')の方向に従い，結合しているリストを介してリングを次々とたどるものと見なすことができる．この例に対する有向グラフを次に示す．これを見ると，記号定義や擬似命令の評価のためにエントリが最終的には使用されてゆく過程を理解することができる．しかし，これは，これらのエントリが最初作られてゆく過程をあらわしてはいない．



この有向グラフの解釈は，次の通りである．

1. 定義アルゴリズムは，標準エントリ P_0 から処理を開始し，主ロケーションカウンタのエントリ L_0 にゆき，カウンタの値を0に設定する．
2. I_1, I_2 のエントリに示されているように，規定の順序に並んだ2つのラベル付き命令が主ロケーションカウンタの制御の下にある．これらのエントリの添字から，これらの命令は，入力プログラム内で2番目および3番目にあらわれたラベル付き命令であることがわかる．
3. 主ロケーションカウンタの下には，これ以外の命令はない．
4. 新しいロケーションカウンタが呼び出され，L_1 とあらわされる．このカウンタに対する初期値を与える擬似命令はないので，この初期値は，先行するロケーションカウンタの最終値と等しいと決める．この場合，L_0 によって決まる．
5. データあるいは記号生成の擬似命令 P_1 と，ラベル付き命令 I_0 とが，

ロケーションカウンタ L_1 の制御の下にある.

6. もう一つのロケーションカウンタのエントリ L_2 が, P_2 によって始められる. この新しいロケーションカウンタの初期値は, 先行するロケーションカウンタの最終値によって決まるのではない. そのかわり, 同じ擬似命令 P_2 によって与えられる.
7. このロケーションカウンタの制御の下には, 只一つのラベル付き命令すなわち I_3 があるだけである. ここで, リングのリストは終る.
8. 定義アルゴリズムは, 別のリストの追跡のために, P-ディクショナリ内の次のエントリを選び出す.
9. P_3 からたどれる命令のエントリは, I_4 だけである. 2番目のリストは終り, これ以上, リストは存在しない.

この説明からわかるように, 擬似命令には2種類ある. リング上にあるものは, データや記号の生成のためのものであり, リスト上にあるものは, ロケーションカウンタやその値を設定するものである.

残念ながら, 有向グラフによる見方では, 有向リングやリストの最終版を作るために必要な反復の回数に関する情報は, ぼやけてしまう. このために, この例題を再び使って, 意味処理を適用し, 異った瞬間における処理のスナップショットを求めてみよう.

1.4.3.3 3段階の処理

プログラム内の記号を定義するために, アセンブラは3段階に分けて仕事をする. 第1段階では, ディクショナリを設定し, 初期値を与える. 第2段階の終りには, プログラム内の命令を順次に走査し終る(これは, 通常“2-パス”アセンブラの第1パスという). このパスの間に, ディクショナリのエントリが作られ, 内容が更新される. この走査の中のすべてのラベル付き命令(擬似または記号命令)に対して, 命令ディクショナリ, すなわち, I-ディクショナリ内にエントリが作られる. 同様にして, 遅延型あるいはロケーションカウンタの初期設定を行なうすべての擬似命令の型に対して, 擬似命令ディクショナリ, すなわち, P-ディクショナリ内にエントリが作られる. ロケーションカウンタに名前をつける擬似命令があるときは, ロケーションカウンタ・ディクショナリ, すなわち, L-ディクショナリにエントリが作られる. この処理の最後の段階では, ディクショナリの記号に対して, すべてロケーションを割り当てる.

段階　I

適当な擬似命令を用いてプログラムの中で特に指定していない限り，アセンブラは，常に，2つのエントリ，すなわち，*L*-ディクショナリと*P*-ディクショナリへのエントリを作って，それらの初期設定を行なう．*L*-ディクショナリのエントリは，プログラムが主ロケーションカウンタの制限の下にあることを示す．*P*-ディクショナリのエントリにより，ロケーションカウンタの値は零となる．

段階　II

この段階では，処理している命令の型を決定しなければならない．遅延型の擬似命令でないラベルなし命令およびロケーションカウンタ型を見つけて，命令の長さを計算する．したがって，これらの命令を処理している間に，*I*および*P*-ディクショナリのエントリ中のパラメータdに初期値が与えられたり，変更されたりする．一方，上述した型の擬似命令を除いたラベルつき命令は，*I*-ディクショナリのエントリを作成することになるので，注意深く扱わなければならない．前にも学んだように，このエントリは，同一のロケーションカウンタをもつエントリとつなげて，リングを作らなければならない．すなわち，*I*および*L*-ディクショナリの適当なエントリに対するポインタを作り出したり変更したりすることを，注意して行なわなければならない．ロケーションカウンタに名前をつけたり，その内容を変更したりする擬似命令に対しては，*P*および*L*-ディクショナリを用いて，それらのエントリを追加したり修正したりする．最後に，遅延型の擬似命令がある．これらの命令は，*P*-ディクショナリのエントリとして退避しておき，あとで解釈する．また，それらがロケーションカウンタと関係しているならば，*L*-ディクショナリのエントリへのポインタL_jを作っておかねばならない．もしも，それらが記号やデータの生成に関係しているのであれば，*I*-ディクショナリのエントリへのポインタI_iを作らねばならない．前者の場合，ポインタはリストを作り，後者の場合，ポインタは，リングを保持する．

段階　III

定義アルゴリズムと呼ぶ反復アルゴリズムを，ここに述べる．アセンブラは，このアルゴリズムにより，ロケーションカウンタのエントリのリストと，個々のロケーションカウンタの制御の下にある命令エントリのリングとを，系統的にたどることができる．

ステップ 1. アセンブラは，P–ディクショナリの最初のエントリから始めて，エントリ中の t に格納してある擬似命令の評価を試みる．それが評価できたり，ロケーションカウンタの値がわかるならば，エントリのポインタに示してあるように，(L_j の)リストか(I_i および P_k の)リングをたどる．一つのエントリは，その前のエントリの値との距離を追加したときに，**定義された**という．これは，リングが閉じるか，または，別の擬似命令が出現したときに終了する．後者の場合，ロケーションカウンタの値が記録される．

ステップ 2. エントリが評価できなかった場合（すなわち遅延型擬似命令）か，リングの追跡が終了した場合，アセンブラは，P–ディクショナリ内の次のエントリを見つけ，ステップ **1.** に述べたように進む．

ステップ 3. 最後に，アセンブラは，P–ディクショナリ内のすべてのエントリを走査しなければならない．そこで，**2** つの可能性がある．

(**a**) すべてのエントリが定義されてしまった．
(**b**) あるエントリが定義されていない．この場合，アセンブラは，以前にリングをたどってしまい定義された擬似命令のエントリは無視しながら，P–ディクショナリのエントリの再処理をする．

ステップ 4. アルゴリズムは，次の **2** つのうち，どちらかで終了する．すなわち，ステップ **3.** (**a**) で，すべての定義ができた場合，および，ステップ **3.** (**b**) で，ある擬似命令の定義が済んでいない場合である．

後者の場合，プログラムの中には，循環的な定義による誤りが存在するに違いない．

1.4.3.4 第 2 パス：定義の代入と式の評価

ディクショナリの内のエントリをすべて定義してしまうと，すべてのラベルつき命令に対するエントリが I–ディクショナリにあるので，アセンブラは，記号ラベルの定義を使える．これらの定義を利用して，アセンブラは全プログラムをもう一度走査して，命令のオペランド中にあらわれる記号に代入し式を評価することができる（このパスは，**2** パスアセンブリの第 **2** のパスになる）．

式の評価においては，アセンブラは，式の値が命令の位置によって影響されるかどうかを決定しなければならない．影響を受ける場合，式は**相対的** (relative) であり，そうでなければ，**絶対的** (absolute) である．あきらかに，絶対式は，定数となるのに対し，相対式は，プログラム（あるいはコントロールセ

クション）の先頭からの相対的な位置になる．相対式および絶対式の算術的結合を決定する値は，次の通りである．

1. 2つの絶対式の和，差，積および商は，絶対式である．
2. 2つの相対式の差は，絶対式である．
3. 相対式と絶対式の和あるいは差は，相対式である．
4. それ以外の結合は，許さない．

ここで，もう一度，次の例を思い出そう．

ラベル	オペレーション	オペランド	コメント
	USE	H	ロケーションカウンタHを使う．
	TRA	Y	
Z	BES	N	N語の記憶ブロックを確保する． (N+1)語目をZとラベルづける．
	BEGIN	M,Z	Zの値を，ロケーションカウンタMへ割り当てる．
	USE		主(空白)ロケーションカウンタを使う．
X	DEC	1,2	2個の10進数1と2とを定義する．
	CLA	X	
W	STO	Z − 1	
	USE	M	ロケーションカウンタMを使う．
Y	NOP	0	
N	EQU	W − X	Nの値は，(W−X)の値と等しい．
	END		

段階ⅠおよびⅡでとられたステップを，**図1.4**，**図1.5**，**図1.6**に示したスナップショットにより説明しよう．

時　刻	*P*-ディクショナリ	*L*-ディクショナリ	*I*-ディクショナリ
t_0	P_0 (0, L_0, BEGIN, Ø)	L_0 (1, , ,) （主ロケーションカウンタに対し）	
段階Ⅰの終りには，リストが作られる			
t_1	P_0 (0, L_0, BEGIN, Ø)	L_0 (1, , , L_1) L_1 (0, , ,) Hに対し	
入力プログラムの第一の文を処理した．			

時 刻	P-ディクショナリ	L-ディクショナリ	I-ディクショナリ
t_2	P_0 (0, L_0, BEGIN, Ø) P_1 (, I_0, BES N)	L_0 (1, , , L_1) L_1 (0, P_1, ,) Hに対し	I_0 (, L_1) Zに対し
最初の3個の文を処理した			

説　明

———————リングの径路　………………リストの径路

1, P_0 で始まり L_1 で終るリストが作られる.
2. L_1, P_1, I_0 よりリングが作られる.
3. エントリ中の要素がこの時点でわからない場合は，そのエントリの中を空白にしておく.

図 1.4 スナップショット(1)

時 刻	P-ディクショナリ	L-ディクショナリ	I-ディクショナリ
t_3	P_0 (0, L_0, BEGIN, Ø) P_1 (, I_0, BES N) P_2 (, , BEGIN M, Z)	L_0 (1, , , L_1) L_1 (0, P_1, ,) Hに対し	I_0 (, L_1) Zに対し
t_4	P_0 (0, L_0, BEGIN, Ø) P_1 (, I_0, BES N) P_2 (, , BEGIN M, Z)	L_0 (1, I_1, , L_1) L_1 (0, P_1, ,) Hに対し	I_0 (, L_1) Zに対し I_1 (0, L_0) Xに対し
最初の4個の文を処理した			
t_5	P_0 ( ) P_1 ( ) P_2 ( )	L_0 (1, I_1, , L_1) L_1 (0, P_1, ,) Hに対して	I_0 (, L_1) Zに対し I_1 (0, I_2) Xに対し I_2 (3, L_0) Wに対し
t_6	P_0 (0, L_0, BEGIN, Ø) P_1 (, I_0, BES N) P_2 (, L_2, BEGIN M, Z)	L_0 (1, I_1, , L_1) L_1 (0, P_1, P_2,) Hに対し L_2 (1, , ,) Mに対し	I_0 (, L_1) Zに対し I_1 (0, I_2) Xに対し I_2 (3, L_0) Wに対し

説　明

────── リングの径路　……… リストの径路

1. すべてのリングやリストが示されているわけではない．
2. t_4 におけるリング L_0 と I_1 とは，t_5 において新しく処理するエントリ I_2 により拡張されている．I_1 におけるポインタが，異なった時刻で異なっているのは，この拡張をあらわしている．

図 1.5　スナップショット(2)

時　刻	P-ディクショナリ	L-ディクショナリ	I-ディクショナリ
t_7	P_0 (　　　　　　　　) P_1 (　　　　　　　　) P_2 (　　　　　　　　)	L_0 (1, I_1, , L_1) L_1 (0, P_1, P_2,) H に対し L_2 (1, I_3, ,) M に対し	I_0 (　　) Z に対し I_1 (　　) X に対し I_2 (　　) W に対し I_3 (0, L_2) Y に対し
t_8	P_0 (0, L_0, BEGIN, Ø) P_1 (, I_0, BES N) P_2 (, L_2, BEGIN M, Z) P_3 (0, I_4, EQU W − X)	L_0 (1, I_1, , L_1) L_1 (0, P_1, P_2,) H に対し L_2 (1, I_3, , 0) M に対し	I_0 (, L_1) Z に対し I_1 (0, I_2) X に対し I_2 (3, L_0) W に対し I_3 (0, L_2) Y に対し I_4 (, 0) N に対し
段階IIが終了			

説　明

────── リングの径路　……… リストの径路

1. P_0 から始めて，エントリの中のポインタをたどると，P_0, L_0, L_1, P_2, L_2 というリストができる．
2. 各ロケーションカウンタのエントリに対して，リングができる．したがって，L_0 に対するリングは，I_1 と I_2 から成る．同様に，ポインタを使うと，L_1 に対するリングは，P_1 と I_0 から成る．L_2 に対するリングは I_3 だけから作られる．
3. エントリ中の空白は，その要素が未知か，あるいは，論理的に一貫していないことを示す．

図 1.6　スナップショット(3)

段階IIIにおけるステップを，次に示す．

1. P-ディクショナリの最初のエントリ P_0 から始める．d を距離の定数とし，$d=0$ とする．

L_0 で　　$d=0$

I_1 で　　$X=d+0=0$

I_2 で　$W = d + 3 = 3$

L_0 で　$d = d + 1 = 4$

L_1 で　$d = 4$

P_1 で　$d = d + (\ \ ) = ?$

N は未だ定義されていないので，アセンブラは，擬似命令 BES を評価できない．したがって，P_1 で処理が終る．

2. P-ディクショナリの次のエントリ P_2 を処理する．Z は未だ定義されていないので，そこで処理が終る．

3. P-ディクショナリの次のエントリ P_3 を処理する．
擬似命令 EQU を，W と X とを用いて評価する．W と X は，前の処理で定義されているので，
I_4 で　$\mathrm{N} = \mathrm{W} - \mathrm{X} = 3 - 0 = 3$
が得られる．

4. I_4 でのエントリは，他のエントリへとつながらず，また，P-ディクショナリには処理すべき新しいエントリも残っていないので，アセンブラは，未だ完全に処理し終っていない P-ディクショナリの最初のエントリから始める．

5. P_1 のエントリを処理する．
擬似命令 BES N は，N が既に定義済みなので，今や評価できる．この命令を評価すると，次のように d を更新することになる．
$d = d + \mathrm{N} + 1 = 4 + 3 + 1 = 8$
したがって，
I_0 で　$z = d + 0 = 8.$
この処理は，L_1 で終了する．

6. P_2 のエントリを処理する．
擬似命令 BEGIN M, Z により，d の値を Z の値と等しくさせる．
すなわち，
$d = z = 8.$
したがって，
I_3 で　$Y = d + 0 = 8,$
L_2 で　$d = 8 + 1 = 9.$

7. 記号の定義は，誤りなく完了し，ディクショナリのすべてのエントリの更新ができた．ディクショナリの最終版を，図 1.7 に示す．

時 刻	*P*-ディクショナリ	*L*-ディクショナリ	*I*-ディクショナリ
t_9	P_0 (0, L_0, BEGIN, Ø) P_1 (, I_0, BES N) P_2 (, L_2, BEGIN M, Z) P_3 (0, I_4, EQU W − X)	L_0 (1, I_1, , L_1) L_1 (0, P_1, P_2,) L_2 (1, I_3, , 0)	I_0 (8, L_1) for Z I_1 (0, I_2) for X I_2 (3, L_0) for W I_3 (8, L_2) for Y I_4 (3, 0) for N

図 1.7 段階Ⅲの終了状態

1.4.4 目的プログラム生成の過程

意味処理が終了すると，この過程では，入力プログラムに対し，目的モジュールあるいはプログラムを生成する．

前にも述べたように，あるコントロールセクションの中で参照する他のコントロールセクション内の記号ラベル（すなわちエントリポイント）と記号名（すなわち外部名）とがある．このようなエントリポイントと外部名とは，そのコントロールセクションをアセンブルしているときには他のコントロールセクションは参照できないので，アセンブラによって定義することはできない．エントリポイント，外部参照名，コントロールセクションの名前に関する情報は，（前に 1.3.3.2 節で述べた）外部記号ディクショナリとして目的プログラムの中に保持しておく．

プログラム内の内部記号は，アセンブラが実現している定義アルゴリズムによって定義される．前に述べたように，記号の定義とは，それを制御しているロケーションカウンタの現在値である．ロケーションカウンタは，命令の群の先頭に対して**相対的**な現在処理している命令の位置を保っているので，一般に，記号の定義は，**相対番地**となる．したがって，命令のオペランドにあらわれる記号の相対的（あるいは絶対的）式は，相対番地（あるいは定数）としてアセンブルされる．記号式が相対的か絶対的かどうかを決めるのに使う規則は，1.4.3.3 で説明した．単純に，相対式は相対番地に，絶対式は絶対番地になるのである．さらに，記号の定義は，番地操作のためのデータとして用いることもできる．これは，データ定義型の擬似命令の中で記号を指定すればできる．たとえば，次の命令

ラベル	オペレーション	オペランド
ADDR	DC	A (LC1)

により，記号名 LC1 を指している ADDR の内容をプログラマは操作することができる(1.3.3.3 節のデータ定義擬似命令を参照)．この擬似命令を評価するために，アセンブラは，その記号を見つけ，相対番地を生成する．相対的な番地になる定義と，定数になる定義とを区別するために，アセンブラは目的モジュールの一部としてディクショナリを作る．これは，**内部再配置用記号ディクショナリ** (internal relocation symbol dictionary) とよぶが，各相対番地に対して，モジュール内のその場所を示すエントリをもっている．

一般に，目的モジュールは，次の2つの部分からできている．

1. 記号ディクショナリ
   a) 外部記号ディクショナリ
   b) 内部再配置用記号ディクショナリ
2. プログラム自身
   a) アセンブルされた命令とデータ

1.4.5 結　論

興味深いことに，前の節に述べた例と等価な IBM 360/370 風のアセンブリ・プログラムを書くことが可能である．練習のために，読者は，新しく書いたプログラムの記号の定義と擬似命令の評価のために，前の節に述べてきたものと同じ意味処理と定義アルゴリズムとを使えることができることを自分で確かめてみるとよい．この練習問題と前述した説明との唯一の違いは，これらのプログラムの記号の定義と式の値とが，同じ数値をとらないということである．これは，IBM 360/370 システムがバイトに基づいた番地方式を採っているからである．それにもかかわらず，新しいプログラムは，今まで用いてきたアルゴリズムによって，正しくアセンブルできる．

しかし，IBM 360/370 アセンブラでこのプログラムをアセンブルすると，次の5個の誤りが生ずる．

1. 飛び越し命令

```
B    Y
```

等価なIBM 360/370風のプログラム			1.4.4節に述べた例		
ラベル	オペレーション	オペランド	ラベル	オペレーション	オペランド
	START				
	USING	*,15			
H	CSECT			USE	H
	B	Y		TRA	Y
	DS	(N)F	Z	BES	N
Z	EQU	*			
M	CSECT				
	ORG	Z		BEGIN	M,Z
	CSECT			USE	
X	DC	2F'1,2'	X	DEC	1,2
	L	1,X		CLA	X
W	ST	1,Z − 4	W	STO	Z − 1
M	CSECT			USE	M
Y	NOP	0	Y	NOP	0
N	EQU	W − X	N	EQU	W − X
	END				

の誤りが示すように，記号番地Yは，現在のコントロールセクションの外側の場所を参照している．

2. 記憶定義の命令

DS　(N)F

の誤りは，全語(4バイト)の記憶を確保する数Nは，未だ定義されていないために生ずる．

3. 同じ記憶定義の命令に対するもう一つの誤りは，重複の数Nを使用することである．かっこのついた式を使用することは違反である．

4. ロケーションカウンタ設定命令

ORG　Z

は，Zが記号ではないので，誤りになる．

5. 格納命令

ST　1, Z—4

の誤りは，Zが他のコントロールセクション内の記号ラベルなので，式(Z − 4)が評価できないからである．

3. 以外の誤りは，すべて，定義が遅れている記号を使用することが直接の原因になっている．3. の誤りの場合，正しい重復の値は，記号でなく定数でなければならない．この制限により，勿論，かっこの使用が禁じられる．定数を用いなければならないので，遅延型定義の記号を用いることは許されない．

IBM 360/370 アセンブラは，ロケーションカウンタ設定の命令の使用の制限，重復因子の制限を行ない，また，他のコントロールセクション内の記号参照の禁止をしている．これらの制限をすれば，アセンブラは遅延型記号定義を扱わないでも済む．したがって，アセンブラが入力プログラムを走査するためのパスの数は，2つに減少できる．もっと詳しく述べるならば，反復型定義アルゴリズムを用いたとき，(**1.4.3.3** 節のアルゴリズムのステップ **3(b)** に示したように)最初の走査の後で *P*-ディクショナリのエントリを再処理する必要がなくなる．最初の走査と，"第2"のパスとにより，IBM 360/370 アセンブラは，実際に2パスのアセンブラとして作られている．

この節で学んだことは，次の通りである．

1. 遅延型記号定義を扱うアセンブラの柔軟さは，定義アルゴリズムの反復の仕方と関係がある．
2. 2パスアセンブラは，定義アルゴリズムを反復して使ってはいない．
3. 反復型定義アルゴリズムを用いなければ，アセンブラは遅延型記号定義の処理ができない．したがって，アセンブラのプログラマは，記号参照の自由を制限され，遅延型の擬似命令を用いることを禁止される．
4. 一方，多重パスアセンブラに対し，2パスアセンブラでは，*P*-ディクショナリの規模が縮少され，単純な1走査の定義アルゴリズムで済むという利点がある．*P*-ディクショナリが小さければ，解釈が少くて済み，1走査ならば再処理がいらないので，時間を節約することができる．

ここに述べた抽象的アセンブラで，アセンブラの速度と，アセンブリ言語の豊かさとの2者関係の特徴づけができたことと思う．このような特徴づけにより，システムプログラマは，アセンブリ言語とアセンブリ・システムとの設計における取引を評価できるようにしてほしい．

1.5 リンキングローダ

アセンブリ言語に擬似命令がない場合，単一のコントロールセクションから得られた目的モジュールを，ローダへの入力として使うことができる．そこで，ローダは，出力として**ロードモジュール**(load module)を作り出し，それを実行のために主記憶に入れることができる．少し追加をすれば，ローダは，独立にアセンブルされた複数個のモジュールを入力として受け取り，結合したロードモジュールを出力として作り出すように拡張することができる．

この型のローダはかなり役に立つけれども，直接参照をしてモジュール間の通信をする機能はもっていない．

EXTRN と ENTRY のようなある種の擬似命令を特別に設計して，コントロールセクション間の通信を組み込めば，一つのコントロールセクションが，他のコントロールセクションを記号的に参照することができる．その際の相互参照を解決するために，再びローダの機能を拡張して，独立にアセンブルし相互に参照し合う目的モジュールを複数個受けつけ，ロードのために記号名を解決するようにする．

ここでは，そのように拡張したローダを，**リンキングローダ**(linking loader)と名付けよう．リンキングローダの主な仕事は，次のようになる．

1. 各目的モジュールのプログラム自身が働く位置を決める．
2. 外部記号ディクショナリを処理しそれを取り除く．
3. データや命令の相対的番地をすべて絶対形式に翻訳し，内部再配置記号ディクショナリを取り除く．
4. チャネル，入力および出力の装置割り当てに関するある種の決定をする．
5. バッファの数，位置および割り当てに関するある種の決定をする．
6. プログラムがうまく実行できるように，主記憶内にデータや命令の最初の配置を行なう．

機能的には，**1.**，**2.**，**3.** の仕事は，プログラムの再配置と記憶の利用に関すること，**4.** と **5.** の仕事は，入力および出力装置の利用に関すること，**6.** の仕事は，ロードに関することである．再配置および記憶の利用に関するリンキングローダの機能は，1.5.1 節で詳しく説明しよう．装置の利用に関するリンキングローダの機能の説明のためには，入力および出力の操作に関する知識が必要なので，入力および出力の操作に関する章で述べる．リンキングローダの

ロードに関する機能については，直接説明することはしない．

これらの基本的な機能のほかに，リンキングローダは，いくつかの新しい能力ももっていてよい．そのような能力の一つとして，**チェイニング**(chaining)があり，これは，小さい記憶空間の中にプログラムを多数集めたものをオーバレイできるようにするものである．別の能力として，**動的ローディング**があり，これは，呼び出しプログラムを実行中に呼び出したサブルーチンをロードできるようにするものである．リンキングローダのこれら2つの追加機能は，次節で述べよう．

1.5.1 再配置と記憶の利用

前節の1.，2.，3.の仕事をみると，リンキングローダは，すべてのプログラム内の参照(たとえば，相対番地，外部記号，内部再配置記号やエントリ)を，その絶対形式に翻訳する処理をすることがわかる．絶対形式とは，CPU が解読できる形式を意味する．そのような過程を**再配置**(relocation)という．あきらかに，アセンブラでなくリンキングローダが，再配置を行なう．

絶対形式が主記憶の番地であるならば，この過程を，**ロード時の再配置**(relocation-at-load-time)という．絶対形式が主記憶以外の番地空間の番地であるならば，その過程を**実行時の再配置**(relocation-at-execution-time)という．この場合，絶対形式を，しばしば**仮想番地**と呼び，その番地空間を**仮想空間**と呼ぶ．仮想番地と仮想空間については，仮想記憶システムの章で説明しよう．ここでは，ロード時の再配置の場合だけにしぼろう．

主記憶は比較的高価であるので，それを上手に利用することに多くの注意を払ってきた．リンキングローダの再配置の能力は，主記憶の効果的な使用を大きく促進する．

プログラムを適当に構成しリンキングローダを利用すれば，使用できる主記憶よりも大きいプログラムまで，主記憶を効率よく使うことができる．

ここでは，記憶の利用を高めるようないくつかの方法を述べよう．

1.5.1.1 チェイニングとオーバーレイ

これは，多分，主記憶の利用に対する最も古典的な方法であろう．この方法では，プログラマは，そのプログラムを部分(通常，ルーチンあるいはコントロールセクション)に分割し，その一つを**主部分**(main section)にして指定する．残りの部分は，**従属部分**(dependent section)と呼ぶ．**チェイニング**と呼ぶリンキングローダの機能によれば，実行時の主部分は，従属部分を実行する

ために呼び出せる．主部分を使用可能な主記憶に置き，残りの記憶を従属部分が共用すれば，主部分は，もういらなくなった従属部分を別の従属部分に置き換えることができる．同一領域をプログラムの異った従属部分で使用する技法を，**オーバーレイ**という．

オーバーレイの技法により，大規模なプログラムに対して小さな主記憶をユーザが利用できるようになるけれども，それは多くの欠点をもっている．

1. ユーザは，プログラムを主部分および従属部分に構成しなければならない．
2. ジョブの準備を注意深くして，オペレーティング・システムに，主部分と従属部分との関係，たとえば，部分のデックの入力順序などの情報を伝えておかねばならない．
3. 部分間の参照に制限がつけられる．たとえば，前出した従属部分は，後出する従属部分で定義しているエントリを参照してはいけない．
4. 主記憶が，一度主部分およびその従属部分に割り当てられてしまうと，実行の間ずっとその状態のままである．実行中に部分の規模が縮少するかもしれないけれども，縮少の結果得られる記憶空間の余りは，プログラムの他の部分では利用できない．一方，従属部分がその規模を増やしたときは，その一部分は，主部分によって別の従属部分が持ってこられたにきにオーバーレイしてもよい．
5. プログラムのすべての部分は，ジョブの準備時に提示し，オペレーティング・システムがリンキングローダのロード時のチェイニング機能を利用するために，適正な結合を設定できるようにしなければならない．
6. 主部分だけが，従属部分を呼び出せるだけである．従属部分は，他の従属部分を呼び出すことは許されない．もしそれを行なうと，チェイニング機構の中に退避してある主部分への戻り番地が消されてしまうだろう．

1.5.1.2　動的ローディング

動的ローディング技法は，チェイニングとは異なり，プログラムの部分はジョブの準備時ではなく動的に定義される．換言すれば，プログラムを構成する部分の数は，主部分の必要に応じて変ることができる．さらに，部分は別々にアセンブルされ，互に呼び出すこともできる．動的ローディング技法を達成するためには，リンキングローダは，自分の知っている場所に**記号参照表**(sym-

bolic reference table)を持っていることが必要である．主部分を再配置するときに，ローダは，主部分の各エントリに対して，2つの情報，すなわち，エントリの記号名とエントリの絶対位置とを作り出す．

主部分が従属部分のためにリンキングローダを呼び出すと，記号参照表を捜して，従属部分の外部記号を処理する．その記号が見つかれば，対応する位置の値を再配置のために使う．そうでなければ，外部記号は未定義であり，従属部分のロードは中止となる．従属部分を持ってくるときにリンキングローダと通信するために，通常，マクロ呼び出しが用意されている．マクロ呼び出しのパラメータとして，従属部分の名前，その部分がロードされる絶対位置を与える必要がある．マクロ呼び出しを発すると，割り込み，トラップ，あるいは，プログラム間の飛び越しによって，部分への要求がオペレーティング・システムのもつ要求処理ルーチンへ伝えられる．このルーチンは，今度は，リンキングローダを働かせて，その部分の再配置を行なう．呼び出しが戻ってくると，再配置の終了かロードの中止かどちらかが行なわれたことがわかる．再配置が成功すると，呼び出し側の部分は，通常の方法で従属部分に飛び越しをする．

従属部分が他の従属部分のロードを要求するときは，同様のマクロ呼び出しをすることができる．唯一つの制限として，ロードすべき従属部分の外部記号が，すべて記号表になければならない(この制限を取り除くためには，どの位の費用がかかるだろうか)．

オーバーレイはチェイニングのように，ジョブの準備前にいろいろの従属部分の記憶の配置をしないので，動的な記憶管理という問題が生ずる．従属部分がロードされるときは，呼び出し側の部分は次のこと必ずを考慮しなければならない．

1. 従属部分に対する主記憶が充分あるか．
2. もし充分でなければ，先行の従属部分によって占有されている主記憶を取り上げられるか．

すなわち，オペレーティング・システムかまたは主部分は，次のような管理能力をもっていなければならない．

1. 使用可能な主記憶の追跡，
2. 先行した従属部分によって占有されていた主記憶の取り上げ(この過程は，屑集めといわれている)，
3. 取り上げた領域をつなげて大きな領域にすること．

要約すれば，コントロールセクションの動的結合とロードの解決は，動的記憶管理の問題によって非常に複雑なことになってしまう．仮想空間は，動的記憶管理において重要な役割を果たすので，動的結合とロードの問題は，仮想記憶システムの章でもう一度考えてみよう．

1.6 あ と が き

一般化したアセンブリ・システムは，[15] と [24] を抽象化したものである．特定のアセンブリ言語の議論は，[8]，[9]，[22] からとったものである．リンキングローダの機能の研究は，[6], [13], [18], [20] を素材にしている．演習問題中のハッシュ化アルゴリズムは，[2]，[12]，[17] から集めたものである．表探索，衝突，ハッシュ化アルゴリズムの他の利用法については，読者は，[7]，[16]，[19] を参照するとよい．ロードモジュールおよび目的モジュールの詳しい形式は，ここでは重要ではない．しかし，[18] と [21] にそれらが報告されている．この章では，マクロアセンブリの命令については扱わなかった．著者は，マクロアセンブリの命令は，擬似命令と考えることができると信じている．マクロ展開の処理は，ロケーションカウンタを用いて実現できる．さらに，マクロ命令に対するコード生成の処理は，データと記号生成型の擬似命令の評価に似ている．このため，マクロ処理系は，アセンブラで採用している定義アルゴリズムの一種を用いて設計できる．マクロアセンブラとして使用しているマクロ処理系と高級言語に対するマクロ生成系は，[1]，[4]，[5]，[23] で論じられている．

参 考 文 献

1. P. J. Brown. "The ML/I Macro Processor." *Comm. of the ACM* **10,** 10 (October 1967): 618–623.
2. W. Buchholz. "File Organization and Addressing." *IBM Systems Journal* **2** (January 1963).
3. J. J. Donovan. *Systems Programming.* New York: McGraw-Hill, 1972, Chapters 1–3.
4. I. D. Greenwald. "A Technique for Handling Macro Instructions." *Comm. of the ACM* **2,** 11 (November 1959): 21–22.
5. M. I. Halpern. "Toward a General Processor for Programming Languages." *Comm. of the ACM* **3,** 4 (April 1960): 214–220.
6. R. Hedberg. "Design of an Integrated Programming and Operating System—Part III: The expanded function of the loader." *IBM Systems Journal* **2** (September, December 1963): 298–310.

7. F. A. Hopgood. "A Solution to the Table Overflow Problem for Hash Tables." *The Computer Bulletin* **11** (March 1968): 297.

8. "IBM 7040/7044 Operating System (16/32k)—Macro Assembly Program (MAP) Language," File No. 7040-21, Form C28-6335-2, Minor Revision, 1965.

9. "IBM System/360 Operation System—Assembler Language," File No. S360-21, Form GC28-6514-8m Eighth Edition 1971.

10. J. K. Iliffe. *Basic Machine Principles.* London and New York: MacDonald/American Elsevier, 1968.

11. W. Kent. "Assembler-Language Macroprogramming: A Tutorial Oriented Toward the IBM 360." *Computing Surveys* **1,** 4 (December 1969): 183–196.

12. W. D. Maurer. "An Improved Hash Code for Scatter Storage." *Comm. of the ACM* **11,** 1 (January 1968): 35–38.

13. J. McCarthy, F. Corbató, and M. M. Daggett. "The Linking Segment Subprogram Language and Linking Loader." *Comm. of the ACM* **6,** 7 (July 1963): 391–395.

14. M. D. McIlroy. "Macro Instruction Extensions of Compiler Languages." *Comm. of the ACM* **3,** 4 (April 1960): 214–220.

15. G. H. Mealy. "A Generalized Assembly System," Rand Memorandum RM-3646-PR, The Rand Corp., Santa Monica, California, August 1963, AD 411 835. (Excerpts were included in the book entitled *Programming Systems and Languages,* Edited by Saul Rosen, New York: McGraw-Hill, 1967, pp. 535–559.

16. R. Morris. "Scatter Storage Techniques." *Comm. of the ACM* **11,** 1 (January 1968): 38–44.

17. W. W. Peterson. "Addressing for Random Access Storage." *IBM Journal of Research and Development* **1** (1957): 130–146.

18. L. Presser and J. R. White. "Linkers and Loaders." *Computing Surveys* **4,** 3 (September 1972): 150–167.

19. C. E. Price. "Table Lookup Techniques." *Computing Surveys* **3,** 2 (June 1971): 1–65.

20. "SPECTRA 70—70/35-45-55 Operating System (TOS) Utility Routines," 70/35-302, May 1968, pp. 6-43 through 6-62 on Linkage Editor.

21. "SPECTRA 70 All Systems—System Standards Reference Manual," 70-00-610, June 1967, pp. C-1 through C-7.

22. "SPECTRA 70 Time Sharing Operating System (TSOS)—Assembler Reference Manual." 70-00-625, June 1969.

23. C. Strachey. "A General Purpose Macrogenerator." *Computer Journal* **8,** 3 (October 1965): 225–241.

24. R. R. Talmadge. "Design of an Integrated Programming and Operating System—Part II: The Assembly Program and Its Language." *IBM Systems Journal* **2** (June 1963): 162–179.

25. P. Wegner. *Programming Languages, Information Structures, and Machine Organization.* New York: McGraw-Hill, 1968, Chapter 2, pp. 107–144.

演 習 問 題

1. 次に示す IBM 360/370 の記号命令と擬似命令の部分集合を考えよ.

    L, ST, A, C, D, M, S, CL, MVI, N, O, X, BAL, BC, LA;
    USING, START, END, DC DS, EQU.

これらの命令のオペランドは，定数と記号ラベルだけに制限する．算術式は許さない．DC と DS に対しては，16 進数と記憶空間の全語を用いた指定だけを許す．USING 命令において，＊印は現在の番地をあらわす．記号ラベルは，最大 4 文字とする．

次の 2 つの課題に対し，前述のアセンブリ言語で 5 つの単純なテスト用プログラムを与えよ．これらのプログラムを書くときには，上述のアセンブリ言語の各命令は，少くとも 1 回使うようにしなければならない．各プログラムは，2 回アセンブルしなければならない．最初は，標準的な IBM 360/370 アセンブラで，次には，自分で作ったアセンブラで行なえ．IBM のアセンブラと自分の作ったアセンブラが作り出す 2 進目的モジュールは，形式上違っていても，これらのモジュールは内容的に等価でなければならない．さらに，これらのテスト用プログラムに対して誤りを検出する能力は，ほぼ同じものでなければならない．

a) 上述のアセンブリ言語で書かれたプログラムを処理できる 2 パスアセンブラを設計し実現せよ．
b) 提案したアセンブリ言語に対し，単一パスアセンブラを設計し実現せよ．

2. 1. のアセンブラを拡張して，2 つの新しい擬似命令 EXTERN に ENTRY も含むようにせよ．

3. 次のことをするリンキングローダを設計し実現せよ．

a) 2. の拡張したアセンブラが生成する目的モジュールを受け付ける．
b) モジュール間の相互参照を，外部記号と内部エントリとを用いて解決する．
c) 相互参照が正しく解決できたならば，ロードモジュールを生成する．
d) ロードを中止しなければならないときには，誤りのメッセージを生成する．

ロードモジュールを，実行のためにコア内に持ってくる必要はない．しかし，ロードモジュールとそれに対応する目的モジュールのリストは，出さなければならない．リストの中には，EXTERN と ENTRY 命令をもつ目的モジュールを少くとも2つ入れて，相互参照を説明せよ．対応するロードモジュールは，IBM 360/370 アセンブラによって作られるものと似たものを簡略化せよ．

4. 次の例のプログラムの記号ラベルを定義しなければならない．すなわち，Z, X, W, Y, N の値を示し，これらの記号ラベルが本当に定義できるならば，その値を導き出した方法を説明せよ．そうでなければ，いくつかの記号ラベルが定義できない理由を示せ．

```
        USE     H
        TRA     Y
Z       BES     N
        BEGIN   M, Z
        USE
X       DEC     1, 2
        CLA     X
W       STO     Z-1
        USE     M
Y       NOP     0
N       EQU     W-Y
        END
```

5. 1.4.1 節に概略を示したような字句解析を実行するアセンブリ・システムの一部を設計し実現せよ．

6. 1.4.2 節に概略を示した構文解析を行うアセンブリ・システムの一部を設計し実現せよ．

7. 1.4.3 節に概略を示した意味処理を実行するアセンブリ・システムの一部を設計し実現せよ．ディクショナリや表のスナップショットを作り出す機構を，この処理の中で作らねばならない．

8. 目的モジュールの標準形を提案せよ．次に，1.5 節で概略を示した最初の3つと最後の機能をもつようなリンキングローダを設計し実現せよ．

9. 上記の4つの設計と実現とを結合し，総合的なアセンブリ・システムを作成せよ．

10. 擬似命令のために開発した定義アルゴリズムに基づき，マクロ展開と命令の生成をするアルゴリズムを設計し実現せよ．

11. マクロ展開と命令生成のアルゴリズムを総合的なアセンブリ・システムの中に組み込んで，マクロアセンブリ・システムを作成せよ．

12. IBM 360/370 および 7090/7040 のアセンブリ言語以外のものを使って，1.4.3.2 節に示した例のアセンブリ・プログラムと等価なものを作成せよ．もしも，プログラムのアセンブルに失敗したならば，失敗の原因を指摘せよ．

13. ハッシュ化アルゴリズム．次のハッシュ化アルゴリズムのプログラムを書いて，記号表に対して一様に分布する番地を生成する能力について比較せよ．

プログラムの入力としては，適当に選んだいくつかのアセンブリ・プログラムのリストの中から，500個の異った記号を集めて使用せよ．

プログラムの出力としては，各アルゴリズムに対し130個位の欄をもつヒストグラムを使用せよ．各欄は，起った衝突の回数を示す．標準偏差の計算も必要である．

結果の議論としては，一つのアルゴリズムが他のものとよりもずっと一様に分布する番地を生成できる要因を論じる必要がある．

使用した記号のリストも必要である．（空白文字が記号の大部分を占めるようなことは，してはならない）．

アルゴリズム　1　[17]

8文字の記号を，2文字づつ4個の部分に分割する．
4個の部分を加える．
和を表の大きさで割る．
余りを相対番地として使う．

アルゴリズム　2　[17]

記号の最初の4文字と，同じ記号の残りの4文字とを加える．
和を平方する．
中央の4文字を取り出す．

それらを，表の大きさで割る．
余りを相対番地として使う．

アルゴリズム 3 [2]，[12]

記号の最初の4文字と終りの4文字との“排他論理和”をとる．
表の大きさより小さくそれに一番近い素数で，その文字を割る．
余りを相対番地として使う．

アルゴリズム 4 [H. F. Guckes]

記号の4文字の部分を，それぞれ等価なパックした10進数に変換する．これら2つのパックした10進数をつなげる．
表の大きさより小さくそれに一番近い（パックした10進数形式の）素数で，このつないだ10進数を割る．
余りを，2進数に変換する．
その2進数を相対番地として使う．

14. 順次および2分探索 [C. W. Gear]

a) **順次探索** N 個の異なった記号を使うアセンブリ・プログラムを考え，それぞれ，$(M+1)$回使われるものと仮定する．定義と使用とが一様に分布し，またどの名前も定義される前に使用されることはないと仮定すれば，K 個の名前が定義済みであるような状況では，探索のために $K/2$ 回の比較を行うと予想できる．これは，次の名前が定義されるまでに M 回行なわれるであろうから，全体として，$M(K/2)$回の比較が行われるであろう．

比較の回数の期待値は，$[MN(N+1)]/4$ となることを示せ．

b) **2分探索** 2分探索技法を使うために必要なことは，記号表が整列されていることである．表を作りながら整列しておくことは，すでに存在する情報を移動する必要があるので，遅い処理になる．アセンブリ・プログラムを一回走査して，すべての記号名を集め表は未整列のままにしておくことなら出来る．多重定義および未定義のチェックは行なわない．

アセンブリ・プログラムを再び走査する前に，2分探索技法を用いて表を整列する．この段階では，多重定義および未定義のチェックができる．次の走査で，番地領域の中の記号名を，2分探索技法を用いて捜すことができる．

異った記号の数が N で，おのおの M 回使用されるとすれば，最初の

走査の終りで行なう整列は，$N\log_2 N$ の規模の処理になるであろう．

全体の比較の回数の期待値は，$(M+1)N\log_2 N$ となることを示せ．

c)　大きな N に対して，順次探索と2分探索とを比較し，結果を示せ．

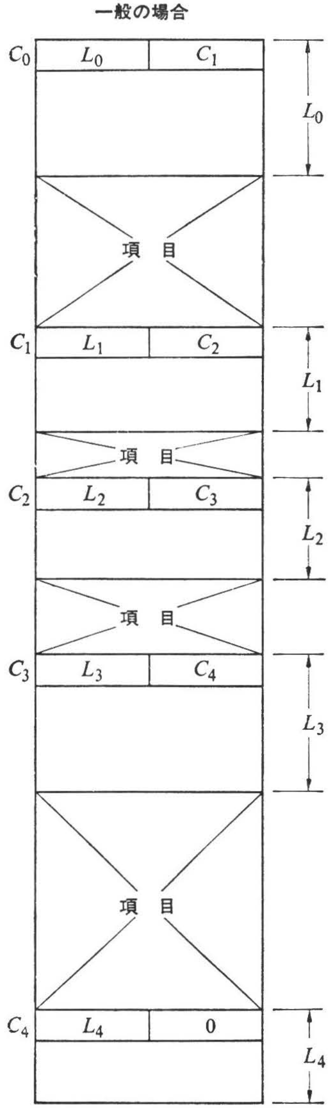


図 1.8

15. 主記憶内の項目を詰めて，使用可能な空間のすべてのブロックを大きな一つのブロックにつなげるような詰めのルーチンを設計し実現せよ．ユーザに割り当てられた主記憶の構成は，図 1.8，1.9，1.10 に示してある．次の

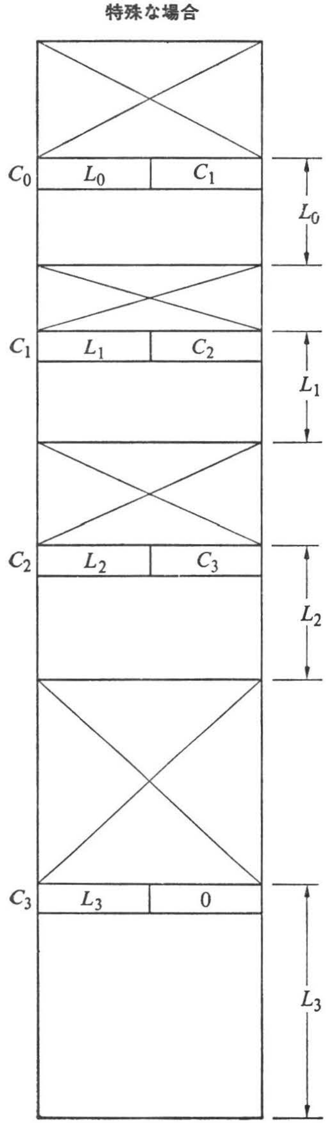


図 1.9

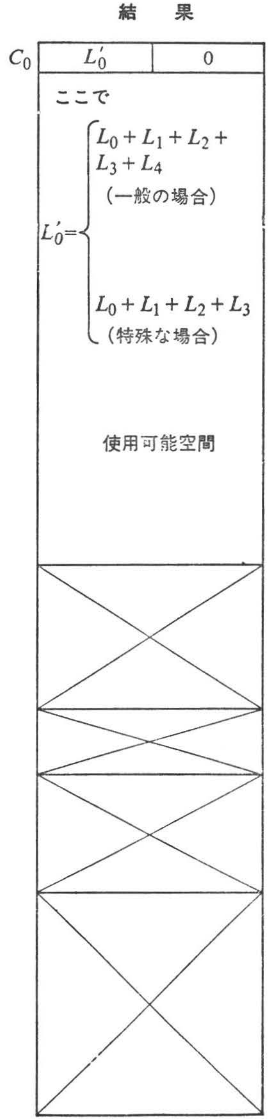


図 1.10

ような定義を用いる.

AVLIS	使用可能な最初のブロックを指すポインタ
ARTOP	主記憶の先端を指すポインタ
L_i	i 番目のブロックの長さ
C_i	i 番目のブロックの番地
項目	番地に依存しないデータとプログラムだけを含むとする.
$i<j$ のとき	$C_i < C_j$

16. 1.4.3.1 節をみればわかるように，エントリ $I_i(d, p)$ と $P_i(d, p, \text{`}t\text{'})$ における距離 d は，i 番目と $(i-1)$ 番目のラベル付き命令の間にある命令の長さの和を意味していた．そこで，次のような特殊な状況を考えよ.

a) $(i-1)$ 番目のラベル付き命令がなければ，どうなるか．すなわち，i 番目のラベル付き命令は，あるロケーションカウンタの制御の下にあるまさに最初のラベル付き命令である．この場合，この距離は，どうやって計算するのか(参考のため，22 ページの最初の例を見よ).

b) ロケーションカウンタが定数または先行するロケーションカウンタの最終値により初期設定されたとき，この計算は複雑になるか.

c) 遅延型の擬似命令にラベルが付いたとき，この擬似命令に対して 2 つのエントリが作られる．一つは，ラベルのために I-ディクショナリの中に，もう一つは，擬似命令のテキストのために P-ディクショナリの中に作られる．2 つのエントリがあるので，2 つの距離が存在する．これらの距離は，どうやって計算するか(ラベル付き擬似命令に対しては，23 ページの 2 番目の例を見よ).

d) 上述の状況を考慮に入れ，これらの特殊な場合も命令の長さが計算できるように，距離 d の定義を拡張せよ.

第 2 章

入出力操作と入出力制御システム

注　意

この章の読むにあたっては，入出力操作，機器，構成に関するハードウェアの特性に関して，[2]，[4]，[11] などに書かれている次のような術語と概念とを知っているものと仮定する．

入出力操作，入出力操作完了フラッグまたは信号，入出力機器，入出力制御装置．
データチャネル．
順次アクセス機器：　テープ．
直接アクセス機器：　ディスク，ディスクアクセス機構，トラック，シリンダ，トラック番地と数，シリンダ番地と数，ドラム，ドラムアクセス機構，セクタ，セクタ番地と数．
CPU，主記憶，記憶サイクル，命令実行時間．

次の術語と概念とは，この章で頻繁に使う．ここで，簡単に説明しておこう．

レコード，論理レコード，物理レコード．
ファイルとファイルラベル．
ブロック，ブロックの大きさ，ブロッキング係数．
バッファ，バッファの大きさ，バッファの切り替え，バッファプール．

記憶媒体に格納されている情報は，通常，**物理レコード**と呼ぶデータの単位に分けられている．物理レコードの大きさは，レコードが存在する媒体に応じて選ばれる．それは，媒体の記憶空間を最適に使い，それに対する全体のアクセス時間を減少させるように設計する．ディスク風(ドラム風)の機器では，ト

ラック(セクタ)またはトラック(セクタ)の一部にアクセスするのに同量の時間がかかるので，最小の大きさは，トラック(セクタ)の大きさである．大きなレコードは，トラック(セクタ)の大きさの倍数になる．そのようなものとして，シリンダ(連続セクタ)の大きさがよく使われる．テープ風の機器では，レコードの大きさはもっと自由に選択できる．一つのレコードを他のものから区別するために，テープ上の物理レコードは，**レコードギャップ**とよぶある固定した空間によって区切られている．レコードギャップを見つけることにより，テープ機器は可変長レコードに対しても正しい入出力操作を始めることができる．しかし，レコードの大きさがたとえばレコードギャップにくらべて小さ過ぎるときには，テープは本来持っているべきものに比べて役に立たない情報しか持たなくなる．さらに，可変長および短いレコードの場合は，固定長および長いレコードの場合に比べて，処理に対する多くの配慮をしなければならない．したがって，直接アクセスの機器にしろ順次アクセスの機器にしろ，汎用システムに対する普通の慣習は，次の通りである．

1. 物理レコードと機器の記憶単位とに一対一の対応がある．
2. 機器上の物理レコードは，同じ長さである．

物理レコードは，主記憶から記憶機器へ写される(または，機器から主記憶へ受け取られる)ので，それを収めるための主記憶が必要になる．物理レコードを写す(受け取る)ための主記憶の空間を，**出力(入力)バッファ**とよぶ．あきらかに，バッファは一時には少くとも一つの物理レコードを収容できなければならない．したがって，バッファの大きさは物理レコードの長さに関係してくる．一方で，物理レコードは比較的遅いアクセス時間と比較的安い機器の記憶空間との兼ねあいをとれるように充分大きくするべきであるのに対し，もう一方では，比較的速く高価なバッファ空間を節約するために充分小さくすべきであるという二者択一をせまられる．バッファ空間を節約する一つの方法は，**バッファプール**を作って，プール内のバッファは，それが使える限り入力にでも出力にでも使えるようにすることである．したがって，特定の操作に一つのバッファを専用として使うことは認められない．バッファの使用可能性と使用の意図とを決めるために，**バッファの切り換え**(buffer switching)を用いる．次に，バッファ切り換え技法について説明する．

2つの固定長のバッファ——一つは入力用もう一つは出力用——を使う単純な場合，問題は入力バッファの中に読み込んで計算をちょうど済ませてある物理レコードの集まりをどうやって機器に書き出すかということである．入力バ

ッファ内のデータを出力バッファに移し，機器に書き出す操作を開始するという解は，明白であるが不経済である．この解が不経済であるのは，記憶から記憶への操作をして，移動の間中 CPU を拘束しておく必要があるからである．そこで，次のような別の解を考えよう．各バッファに対し，次のものからなるエントリを用意する．

1. バッファの状態 S(たとえば "0" は空き，"1" は使用中をあらわす).
2. バッファの長さ L.
3. バッファの主記憶の番地 A.

バッファの使用目的は，これらのエントリの位置によって決まる．この例では，最初のエントリは常に入力バッファを，第二のエントリは出力バッファを指す．**図 2.1** を見ると，番地 A_j から始まるバッファは，最初のエントリが番地 A_j をもっているので入力バッファである．同様に，番地 A_i のバッファは，二番目のエントリが番地 A_i をもっているので，出力バッファである．入力バッファのデータを機器に書き出す準備ができたとき（すなわち，入ってきた入力データの処理が終ったとき），かつ，出力バッファの状態が空きのとき，**図 2.2** に示すように，単にエントリの A_i と A_j とを交換するだけでよい．この交換の後，機器への書き出し操作を開始する．さらに，新しい入力操作が必要な

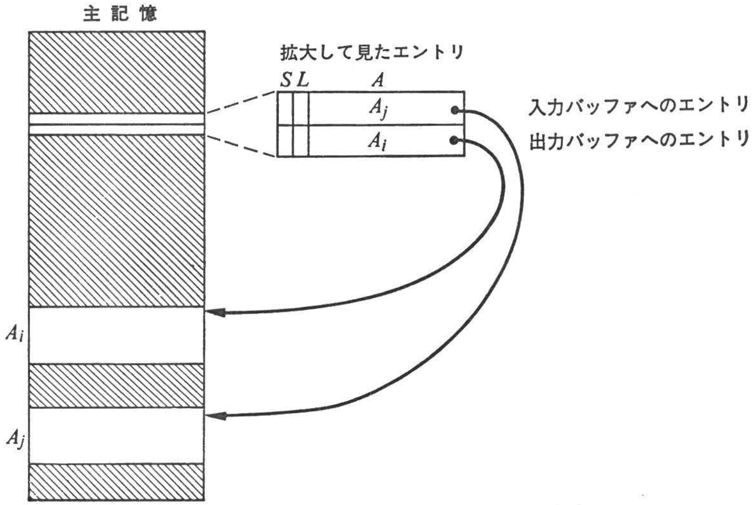


図 2.1　入力/出力バッファ

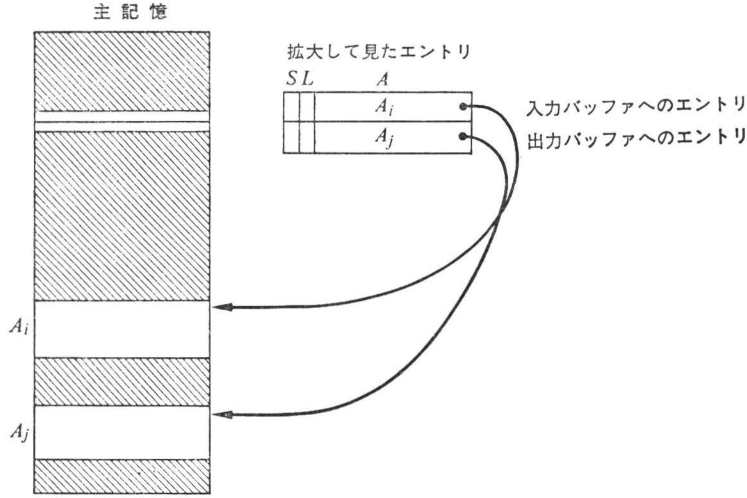


図 2.2　入力/出力バッファの切り替え

らば，入力操作を開始できる．この技法を使えば，記憶から記憶へ移す操作は不必要になるばかりでなく，新しい入力と出力の操作もすぐに開始できる．バッファの番地の交換は，データを一つの番地から別のところへ物理的にでなく論理的に移すことになり，入力バッファを出力バッファとして使えるようにでき，又，その逆も可能である．これが，バッファの切り換えの技法である．

格納と検索に使うユーザの情報の最小単位を，**論理レコード**と呼ぼう．論理レコードの大きさが物理レコードの大きさを意味するわけではないことは，あきらかである．媒体上の記憶空間を最適に使い，最短でアクセスするためには，ユーザの論理レコードを記憶上の物理レコードに変換しなければならない．さらに，物理レコードを媒体から検索するときには，ユーザに論理レコードとして戻さねばならない．論理レコードを物理レコードの大きさをもったバッファ（さらに，制御用の2，3語，すなわち記述子を加えた）中に一度に一つづつ置いて論理レコードを物理レコードに構成していく過程を，**ブロッキング**(blocking)という．バッファが論理レコードをこれ以上収容できなくなったとき，バッファの内容を物理レコードとして媒体上に書き出す．このために，物理レコードを**ブロック**ともいう．検索してきた物理レコードを論理レコードに分解し，ユーザに一つづつ見せる過程を，**デブロッキング**(deblocking)という．

共通の特性，たとえば形式や使い方をもつ論理レコードの集まりを**ファイル**と呼ぶことができる．システムでは，ファイルはブロックの集まりとして格納している．一つのフィイルのブロックを他のファイルのものから区別するために，一意的な**ファイルのラベル**をファイルに割り当てる．

ファイルの**ブロッキング係数**(blocking factor)とは，ファイル内の論理レコードの数とファイルのブロックの数との比と定義する．したがって，ブロッキング係数は，ブロッキングに必要な論理レコードの平均の個数である．

2.1　入出力操作の例

簡単なディスクに書き出すルーチンの論理を，**図2.3**のフローチャートで示す．このルーチンは，ディスクの最初のトラック番号，バッファの番地，および，バッファの大きさを与えれば，バッファ内のデータをディスクの連続したトラック上に書き出すことができる．このルーチンでは，2つの入出力操作を使っている．第1に，トラックをシーク(seek)する命令である．あきらかに，与えられたトラック番号を基礎にして適正なトラックが見つからなければ，この命令の結果は誤りの信号またはフラッグになる．第2に，トラックに書く命令で，これにより，バッファ内のデータを連続したトラック上に書き出す．ここで，次のことがわかる．(1) トラックをシークする命令のCPU実行時間†は，加算や格納などの計算機の命令の実行時間と同じ位であるけれども，ディスクが適正なトラックをシークするのに必要な時間(すなわち，アクセス時間)は，実行時間の数千倍という大きさになる．(2) トラックに書き出す命令の場合，命令の実行時間はやはり小さい．しかし，主記憶内のバッファから連続したトラックへデータを移動するために必要な転送時間は，バッファの大きさと転送率との積になるので，勿論大きなものとなる．

(1) において，アクセス時間の間，正しいトラックが見つけられるのを妨げるような新しい入出力操作を出してはならないことに気がつく．したがって，トラックに書き出す命令を出せる迄には，待ちの期間がある．一方，この期間を入出力命令以外の命令の実行のために使ってはならないという理由は，全然見あたらない．これが本当ならば，CPUは命令を実行するのに忙しく，入出力機器はそのシーク操作を続けることができる．理想的には，シーク操作が完了する(すなわち，トラックが見つかる)ときに，CPUは，バッファを満す仕事も含んだ命令実行の列の論理的な終端にちょうど達していればよい．そうで

† CPUが他の操作ができるように空いてはいない時間を実行時間という．

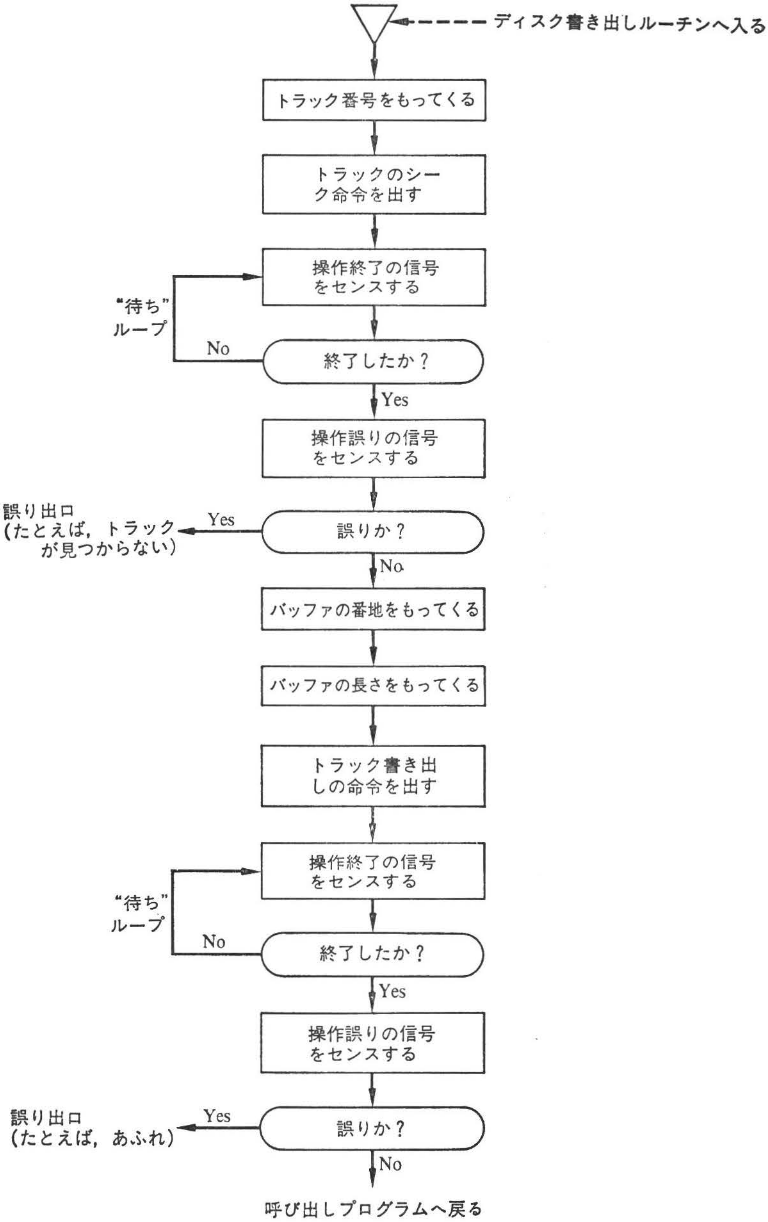


図 2.3　ディスク書き出しルーチン

あれば，トラックは準備ができバッファは満たされているので，トラックに書き出す命令をすぐに実行できる．目的を達成するためには，アクセス時間を，実行時間と重ね合わせるべきである．そのようなCPUと入出力操作の重ね合せのためには，入出力ルーチンを注意深くプログラムしなければならない．

(2)においては，プログラムからディスクにデータを転送している間は，同じ機器に対し新しい入出力命令を出してはならないことに気がつく．そうでなければ，トラックに書き出す命令は，正しく完了できない．長い転送時間のために，別の入出力命令を出す迄，やはり待ちの期間がある．前述した待ちの期間と異なり，この期間は次のことを含んでいる．物理レコードの転送のためにフェッチをするごとに，主記憶バッファへのアクセスがある．一般的な規則として，データの転送のために入出力機器が記憶アクセスを必要とするときは，いつでもCPUはいま実行中のものの中断をしてまでも，入出力機器に記憶アクセスを譲らなければならない．しかし，記憶アクセス時間は，転送時間に比べて短い．入出力機器による記憶アクセスは短く，CPUによる命令実行の中断と再開とは自動的に行なわれるので，やはり，待ちの期間にCPUが非入出力命令を実行させることは，合理的である．このような考察から，アクセス期間中であろうと転送期間中であろうと，CPUと入出力操作との重ね合せは望ましいことであると結論づけてよかろう．ついでに，CPUと入出力操作とを充分に重ね合わせたプログラムを，**釣り合いのとれたプログラム**と名づける．

2.2 バッファの大きさとバッファの割り当て

2.2.1 釣り合いのとれたプログラムを完成できるか

前節の議論から考えると，釣り合いのとれたプログラムはすぐに完成できそうに見える．理解を助けるために，釣り合いのとれたプログラムと，CPUおよび入出力操作の重ね合せの概念を順序づけてもっと完全に扱ってみよう．これは，詳細な例を使って行なう†．読者は，詳細な事項に耐えて読んで欲しい．前と同じように，われわれの目的は，例題の細部を非難するのではなく，むしろ概念をつかむことにある．

最初のものは，語のブロックをディスクに書き出すサブルーチン——第2の

† ここでは，PDP-8のアセンブリ言語で書いたプログラムを使って説明する．しかし，PDP-8のアセンブリ言語に関する知識は必要ない．PDP-8の入出力命令の実行時間は，3.75 μsecである．他の命令の実行時間は，1.5 μsecから4.5 μsecの範囲にある．ミニディスクは，66 μsec/語の転送率と，平均16.67 msecのアクセス時間をもつ．これらについては，次頁の表にまとめてある．

プログラム DSCWT を呼び出す主プログラムである．ここでは，バッファの中に入れる語を作り出しバッファの内容がディスクに書き出されたときには常にバッファを満たすことをする主プログラムの部分については，述べていない．しかし，ここでは，そのような語を作り出すために，平均1命令につき3μsec としておよそ35命令の実行が必要であると仮定しよう．さらに，これらの語は，1000番地にある200語の大きさのバッファに格納される．この説明図では，バッファは示していない．

サブルーチン DSCWT は，次の情報が適正に与えられたとき，語のブロックをディスク上に書き出すことができる．

1. ディスク上に書き出す最初の語の主記憶の番地，
2. 連続して書き出す語の数，
3. トラックの番地とトラックの数，
4. 誤りおよび正常の場合の戻り番地．

ディスクに転送される語数は，数千のオーダである．**図2.4**に示した DSCWT ルーチンは，2.1節に示したプログラムの論理に似ている．

ラベル	オペレーション または定数	オペランド	コメント
MAIN	JMS	DSCWT	ディスク書き出しルーチンへ飛べ．
	1000		ディスクに書き出す語の先頭のコアの番地
	700		ディスクのトラック数
	-200		語のカウントの負値
	0		ディスクのトラックの番地
	HLT		ルーチンからの誤りの戻り番地
	HLT		正常の戻り番地
			ディスクの転送率 66 μsec/語
			平均アクセス時間 16.67 msec

次の記法を使おう．

n　　バッファ内の語の数，すなわち，バッファの大きさ，
r_{io}　　1語あたりの入出力装置の転送時間，
t_a　　入出力装置のアクセス時間．

ラベル	オペレーション，オペランドまたは定数	コメント	必要な実行時間（μsec）
DSCWT	Ø	ディスク書き出しルーチンのエントリ	
	CLL CLA CMA		1.5 + 1.5 + 1.5
	TAD I DSCWT		4.5
	DCA I CA	CAに，ディスクに書き出す語の先頭の番地を格納する．	4.5
	ISZ DSCWT		3.0
	TAD I DSCWT	トラック番号をアキュムレータに入れる．	4.5
	DEAL	トラックをシークする．	3.75
	ISZ DSCWT		3.0
	CLA CLL		1.5 + 1.5
	TAD I DSCWT		4.5
	DCA I WC	WCに語のカウントを格納する．	4.5
	ISZ DSCWT		3.0
	TAD I DSCWT	トラックの番地をアキュムレータに入れる．	4.5
	ISZ DSCWT		3.0
	DMAW	主記憶から語をディスクへ書き出す．	3.75
	DFSC	完了フラッグを設定し，もし完了ならば次の命令を飛ばす．	3.75
	JMP.-1		1.5
	DFSE	誤りがあれば，誤りフラッグを設定する．そうでなければ，次の命令を飛ばす．	
	JMP ERR		1.5
	CLA		1.5
	ISZ DSCWT		3.0
	DCMA	完了フラッグを戻す．	3.75
	JMP I DSCWT	呼び出しプログラムへ戻る	3.0
ERR	DEAC		
	JMP I DSCWT		
			72.0
WC	775Ø	語のカウントを格納する一定の主記憶の場所	
CA	7751	番地を格納する一定の主記憶の場所	

図 2.4 DSCWT ルーチン

そこで，n 語の入出力処理時間の総和は，

$$t_{i0} = t_a + nr_{i0} \tag{1}$$

となる．

この例では，次のようになる．

$$\begin{aligned} t_{i0} &= 16{,}670 + (200)\cdot(66) \\ &= 16{,}670 + 13{,}200 = 29{,}870\,\mu\text{sec.} \end{aligned} \tag{2}$$

同様にして，次の記法を使おう．

r_p　　バッファ内の各語に必要な CPU の処理時間の量，

t_p　　入出力ルーチンを実行するのに必要な CPU の処理時間の量．

CPU 処理時間の総和は，

$$t_{\text{CPU}} = t_p + nr_p \tag{3}$$

となる．

例で述べたように，バッファ内に置く語を作り出すために，35命令の実行をするので，

$$r_p = 35 \times 3 = 105\,\mu\text{sec.}$$

したがって，

$$\begin{aligned} t_{\text{CPU}} &= 72 + (200)\cdot(105) \\ &= 21{,}072\,\mu\text{sec.} \end{aligned} \tag{4}$$

(2)と(4)とを比較すると，CPU がバッファを満すことを始めると同時にシークを発したときには，

$$t_{i0} > t_{\text{CPU}}$$

となることがわかる．これは，プログラムが**入出力限界**(I/O–bound)であることを意味している．換言すれば，CPU は，ある時間は空いていることになる．

$t_{i0} = t_{\text{CPU}}$ となる“釣り合いのとれた”プログラムは，入出力装置も CPU も空きにしないと結論づけてもよいだろう．これは，バッファの大きさを調整して，

$$t_{i0} = t_{\text{CPU}} \text{ または } t_a + nr_{i0} = t_p + nr_p$$

の等式が成り立つようにすれば，達成できる．したがって，

$$n = \frac{t_a - t_p}{r_p - r_{i0}} = \frac{16,670 - 72}{105 - 66} \cong 426 \text{ 語}.$$

しかし，“釣り合いのとれた”プログラムは，CPU と入出力装置を完全に使い切るとは保証していない．この理由は，(a) シークが完了し，読み/書きヘッドが正しいトラックの上に置かれ，(b) 出力バッファが完全に満たされる，までディスク上へ書き出すことはできないからである．

バッファの大きさとして 426 語を使うならば，バッファ内に最初に置く 426 語を作り出す前に，ディスクのシーク操作 DEAL を開始できるであろう．ディスクのシーク時間は，

$$16,670\,\mu\text{sec}$$

であり，426 語を作り出すためには，

$$n_r = 426 \times 105 = 44,730\,\mu\text{sec}$$

かかるので，CPU と入出力処理の重ね合せは，16,670 μsec である .残りの

$$(44,730 - 16.670) = 28,060\,\mu\text{sec}$$

は，入出力がひまになる．すなわち，入出力装置は，CPU がバッファを満たすまで，データの転送の開始を待たなければならない．

上の説明からわかるように，“釣り合いのとれた”プログラムは，実は，**計算限界**(computation-bound) となる．

もう一つの合理的な方法は，CPU の時間と入出力装置のアクセス時間とが完全に重なり合うようにバッファの大きさを選ぶことであろう．この場合，バッファの大きさは，

$$n' = \frac{t_a}{r_p} = \frac{16,670}{105} = 159 \text{ 語}$$

となる．したがって，

$$\begin{aligned} t'_{i0} &= t_a + n'r_{i0} \\ &= 16,670 + (159)\cdot(66) \\ &= 16,670 + 10,494 = 27,164\,\mu\text{sec} \qquad (5) \end{aligned}$$

$$\begin{aligned} t'_{\text{CPU}} &= t_p + n'r_p \\ &= 72 + (159)\cdot(105) \end{aligned}$$

$$= 72 + 16,695 = 16,767\,\mu\text{sec} \qquad (6)$$

となる．式の対(2), (4)と(5), (6)とを比較すると，(2)と(4)とからは，

$$\begin{aligned} t_{i0} - t_{\text{CPU}} &= 29,870 - 21,072 \\ &= 8,798\,\mu\text{sec}, \end{aligned}$$

(5)と(6)とからは，

$$\begin{aligned} t'_{i0} - t'_{\text{CPU}} &= 27,164 - 16,767 \\ &= 10,397\,\mu\text{sec} \end{aligned}$$

が得られる．上述のことから，バッファの大きさを159に選んでも，元の大きさ200に比べて改善されていないことがわかる．ディスクのアクセス時間は完全に計算時間と重なり合うけれども，主記憶からディスクへのデータの転送する時間の間は，CPUは一部分空きになる．

一般に，大部分の可動ヘッドディスクのアクセス(すなわち，シーク)時間は異なっている．それは，望んでいるトラックに到達するために読み/書きヘッドが動かねばならない量に比例する．したがって，固定長のバッファを選んだときの最善の方法は，**平均アクセス**時間を重ね合わせることである．実際には，プログラムは，ある実行では入出力限界であり，別の実行では計算限界となるかもしれない．

前述の議論で使った式は，一般化して，レコードやブロックの入出力操作を反映するようにできる．あきらかに，レコードは語にすることができる．しかし，

$$t_{i0} = t_a + nr_{i0} \qquad (7)$$

ただし，　n は，バッファ中のレコードの数，
r_{i0} は，レコードあたりの入出力装置の転送時間，
t_a は，入出力アクセス時間，

が成り立つ．また，

$$t_{\text{CPU}} = t_p + nr_p$$

ただし，　r_p は，バッファ中で各レコードを処理するために必要なCPUの時間，
t_p は，入出力ルーチンを実行するために必要なCPUの時間，

が成り立つ．

t_p は，一定のむだ時間 (overhead) であり，大きな n に対して比較的小さいので，上記の式は，次のように簡単化できる．

$$t_{CPU} = nr_p. \tag{8}$$

式(7)と(8)とは，後の節で使うだろう．

要約すると，入出力アクセス時間，入出力転送率，処理時間，プログラム論理が不統一なので，入出力装置や CPU をプログラムが完全に使い切ることは，不可能でないにしても難しいことである．これは，一つのプログラムが完了するまで他のプログラムを並行的に処理しない環境では，特に顕著になる．

そのような環境では，入出力装置の不活発さを犠牲にしてでも，CPU の利用をできるだけ多くすることに，強い関心が払われる．すなわち，計算と入出力との完全な重ね合せが不可能であるならば，ユーザのプログラムを計算限界にするようにあらゆる努力を払う．

上述の説明からわかるように，プログラムを計算限界にする方法は，大きなバッファを割り当てることである．この考え方により，多くのオペレーティング・システム(たとえば，第1章のアセンブリ・システム)は，システムの入出力制御システム (IOCS) またはリンキングローダを介して，ユーザプログラムのバッファにできるだけ多くの主記憶を割り当てようとしている．したがって，前章のリンキングローダの節では後まわしにしていたが，入出力装置の扱いとバッファの割り当てにおけるリンキングローダの役割を論ずることができる．

一方，制限の少ない環境では，最初のプログラムを完了しなくても複数個のプログラムを処理してよいが，そこでは一つのプログラムの CPU 操作と別のプログラムの入出力操作とをかなり重ね合わせることができる．さらに，入出力機器，チャネル，バッファを追加すれば，一つのプログラムの入出力操作と，別のプログラムの入出力操作とを重ね合わせることさえ可能である．プログラムの間でそのような重ね合せをすることを，**並行処理**(concurrent processing)というが，これは，2.4 節で述べよう．

2.2.2 バッファの割り当てアルゴリズム

バッファが大きければ CPU の利用がふえるので，バッファ割り当てアルゴリズムの目的は，バッファを必要とする入出力操作に対して，出来るだけ多くの主記憶 (すなわち，プログラムの計算の部分がすべてロードされた後に残された主記憶の量) を割り当てることである．

入出力操作に対するバッファの最小の大きさは，通常，入出力操作に対する

ユーザのプログラムの中で指定される．もし指定されていなければ，システムが，入出力装置の特性とか装置上の情報の物理的配置などに基づいて決定する．

以下に，単純なバッファ割り当てアルゴリズムを示すが，そこでは次の事項に基づいて追加のバッファを入出力操作のために割り当てる．

1. 他の入出力操作の頻度と相対的な入出力操作の頻度――高い頻度ならば，入出力処理が多いので，プログラムは入出力限界になるかもしれない．したがって，CPU の利用を向上させるためには，可能であるならば，追加のバッファを高頻度の入出力操作に最初に割り当てるべきである．
2. 追加のバッファを共用するかもしれない入出力操作の数――同一の大きさのバッファをプールすることにより，これらのバッファに付随した入出力操作は，プール内のバッファを共用できる．これは，バッファの大きさを拡大した効果と同じことになる．

一般に，プールに付随している入出力操作の数は，プールごとに変る．バッファを，少数の入出力操作の付随するプールよりも，多数の入出力操作の付随するプールに割り当てた方が，ずっと有効に共用できる．その結果，CPU の利用も向上する．

9個の入出力操作 IO_1，IO_2，…，IO_9 をもつプログラムを考えよう．最小のバッファの大きさと，各入出力操作の相対的な頻度とを，下の表に示す．割り当てに使える主記憶は，この例では，1600語である．

入出力操作	最小のバッファ（語）	使用の相対的頻度
IO_1	200	2
IO_2	200	1
IO_3	100	0
IO_4	100	0
IO_5	100	1
IO_6	50	0
IO_7	50	3
IO_8	50	1
IO_9	100	2

同一の大きさのバッファを一つのプールに入れるために，入出力操作を次のようにまとめる．

プール	バッファの大きさ	入出力操作の数	必要なバッファの数	頻度の和	必要な最小の大きさ
1	200	$2(IO_1, IO_2)$	2	3	400
2	100	$4(IO_3, IO_4, IO_5, IO_9)$	4	3	400
3	50	$3(IO_6, IO_7, IO_8)$	3	4	150

上の図から，最小限必要な記憶は950語であることがわかる．すべてのバッファを2倍にすると，$(2 \times 950) = 1900$ 語必要になる．使用できる主記憶が1900語よりも大きければ，元の記憶の要求を3倍にすることを考える．しかし，この例では，使える記憶は1900語より小さい．したがって，先ず，最小限必要な950語を割り当てる．次に，残りの650語をプールの中に分配することにする．

A_i を，プール i が望む追加バッファの数とし，S_i を入出力操作の頻度とする．

重みを，次のように作る．

$W_i = A_i \times S_i \qquad (i = 1, 2, 3$ に対し).

ステップ 1. 残りの主記憶を，重みが最大のプールへ分配する．

ステップ 2. プールに追加バッファが割り当てられたときには，その A_i を，直ちに1だけ減らす．こうすると，そのプールの重みは，S_i だけ減少する．

これらのステップは，割り当てる主記憶がなくなるまで続ける．すべてのバッファを2倍にしようと考えていたので，望んでいる追加バッファ A_i は，次のとうりである．

$A_1 = 2$

$A_2 = 4$

$A_3 = 3$

使用可能な主記憶	プール1 バッファの大きさ=200			プール2 バッファの大きさ=100			プール3 バッファの大きさ=50			重みによって選んだプール
	A_1	S_1	W_1	A_2	S_2	W_2	A_3	S_3	W_3	
650	2	3	6	4	3	⑫	3	4	12	Pool 2
550	2	3	6	3	3	9	3	4	⑫	Pool 3
500	2	3	6	3	3	⑨	2	4	8	Pool 2
400	2	3	6	2	3	6	2	4	⑧	Pool 3
350	2	3	⑥	2	3	6	1	4	4	Pool 1
150	1	3	3	2	3	⑥	1	4	4	Pool 2
50	1	3	3	1	3	3	1	4	④	Pool 3
0	1	3	3	1	3	3	0	4	0	

要約すると，次のようになる．

プール	バッファの大きさ	入出力操作の数	頻度の和	必要なバッファ数	割り当てられた追加バッファ	バッファの総数
1	200	2	3	2	1	3
2	100	4	3	4	3	7
3	50	3	4	3	3	6

このアルゴリズムでは，入出力の数と使用頻度とを合わせて最大となるプールが，最大の追加バッファを受けている．

一般に，バッファ割り当てアルゴリズムは，バッファの型(たとえば，入力型や出力型)や入出力操作のモード(たとえば，スーパーバイザモードやユーザモード)など，他のいろいろの要因に左右される．

2.3　入出力干渉と入出力機構の利用

2.3.1　記憶サイクルスティーリングの現象

この章の前の部分で述べた例を見ると，1語が主記憶からディスクへ転送されるごとに，転送の開始のために3個の記憶サイクル——すなわち4.5 μsec——が必要であることがわかる．その一つのサイクルは，主記憶から語を取り出すために使う．他のサイクルは，制御語WC(語のカウント)およびCA(取り出す語のコアの番地)を取り出し，更新し，再びしまうために使う．これらの3個のサイクルは，たとえCPUが実行している計算で同時に主記憶にアクセス

る必要が生じても，CPU には使用できないものである．IBM 360/370 システムでは，いろいろの型のチャネルや機器を扱えるので，チャネルコマンドや機器のオーダの内容は工夫して作ってある(2.3.2 節参照)．これらのコマンドやオーダは，通常，**チャネルコマンド語**(channel command word)というデータとして，主記憶の中に格納してある．入出力操作が要求されると，システムは主記憶の指定された位置から適当なチャネルコマンド語を持ってきて，指定されたチャネルや機器に送る．ここでも，記憶から取り出すために，CPU が他の目的に使えないサイクルを必要とする．このような現象を，一般にデータ転送のための**サイクルスティーリング**(cycle stealing)とよぶ．

主記憶へのアクセスを，CPU でなく入出力転送の操作のために与えるのは，データ転送のためのサイクルスティーリングは頻度が少ないと思っているからである．例では，語をディスクに転送するのに 66 μsec かかるので，取られるサイクル時間は，転送時間全体の 15 分の 1 (4.5/66 = 1/15)を越えることはない．

サイクルスティーリングの現象は，データチャネルと入出力装置にハードウェアを追加すると，最小にすることができる．たとえば，例において，ディスク装置とデータチャネルとに 2 つのハードウェアを与える——一つは制御語 WC，もう一つは制御語 CA を格納しておく——ならば，3 個のサイクルのうち 2 個は取る必要がなくなる．何故ならば，これらの制御語は，最初のデータ転送を除いて，データ転送ごとに主記憶から取り出してくる必要がもはやなくなるからである．同様に，IBM 360/370 システムのチャネルコマンド語は，チャネルの局所的記憶に格納することもできる．ディスク，ドラムやテープのような二次記憶機器は，大量のデータを格納するように設計してある．主記憶と二次記憶機器間のデータ転送は，通常，数百語(バイト)あるいは数千語(バイト)になる．この例の 2 つのレジスタ，あるいは，IBM 360/370 システムに提案されている局所的記憶のような何らかのハードウェアを追加すると，数百あるいは数千の記憶サイクルを CPU の記憶参照型の計算のために解放できる．しかし，サイクルスティーリングの最小化は，ハードウェアのレジスタやチャネルの局所的記憶にかかる追加の費用と天秤にかけなければならない．

2.3.2 CPU を入出力操作の一部として使うこと

例において，制御語の取り出しに 2 個の記憶サイクルを使うが，その更新と格納はデータチャネル内に作られたハードウェア機構がやってくれるので，更新と格納のための記憶サイクルは何も必要とはならないことがわかる．ハード

ウェアの機構がなければ，プログラマは，次のことに対する少なくとも6個の命令を与えなければならないだろう．

1. 転送した語の完了フラッグを読むこと，
2. 語のカウントが零になっていなければ，一つ減らすこと，
3. 語のカウントが零のとき，入出力操作を停止すること，
4. コアの番地を1だけ増やすこと，
5. 現在転送されようとしている語を指定したレジスタに置くこと，
6. 入出力命令を実行すること，
7. 完了フラッグをリセットすること．

すなわち，1語を転送するごとに，CPUは，これら6個の命令によって，少くとも$(6)\times(1.5)=9\,\mu$sec間拘束される．

制御語の更新はCPUがプログラムを介して行なうので，データチャネルの中にハードウェア機構を追加する必要はない．しかし，制御語の更新にCPUの論理および算術の機能を使うので，入出力操作により，CPUは，それと無関係な計算の実行ができなくなってしまう．

要約すると，サイクルスティーリングの現象により，記憶へのアクセスを入出力操作に優先的に与えてしまうので，CPUは記憶を参照する命令の実行を妨げられてしまう．CPUを入出力操作の一部として用いると，その操作のためにCPUを拘束する．まとめていうと，入出力操作によって生ずるこのような遅れを，**入出力干渉**(I/O-interference)という．

2.3.3 データチャネル

論理的には，データチャネルは，入出力機器とCPUの制御の下にある主記憶との間に置かれた衛星計算機と考えることができる．

チャネルは，その容量に応じて，自分自身の記憶レジスタ，算術および制御

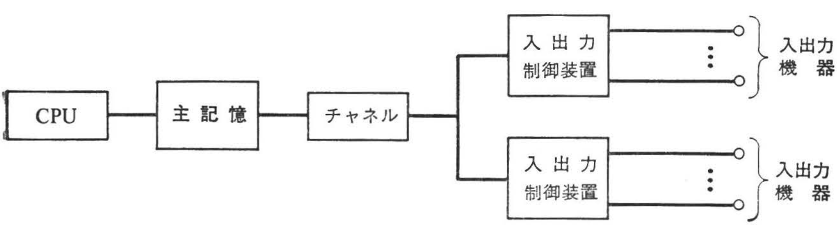


図 2.5　単一チャネル計算機システム

装置をもつか，または，CPU の主記憶と論理機構とを共用している．自分の記憶や論理機構をもったデータチャネルは高価なものであり，ある構成においては，CPU よりも高くつくこともある．したがって，専用の記憶や論理機構をデータチャネルにもたせるのは，経済的ではない．

一方，CPU の主記憶と論理機構とを共用するとき，入出力干渉の問題が生じてくる．したがって，システム構成と経済とが許すならば，自分自身の記憶と論理機構とをもつデータチャネルを使う方が望ましい．この場合，CPU を拘束し空きにしてしまうことを最小にするように努力することができる．したがって，データチャネルを使えば，CPU は次のことを遂行できる．

1. 最小の記憶アクセスと処理時間とで，入出力操作を開始する．
2. 最小の干渉で，計算と入出力のデータ転送とを並行的に進める．

チャネルによって実行する命令を，**チャネルコマンド**という．チャネルコマンドのプログラムは，チャネルの記憶かまたは主記憶の中に置く．後者の場合，CPU はチャネルコマンドを適当なチャネルに送って，そのチャネルで実行させる．基本的なコマンドの型は，次の通りである．

1. 読み込む———データを入出力機器から主記憶へ転送させる．
2. 書き出す．
3. センスする——入出力機器に機器に依存した状態の情報を要求する．
4. 制御する———入出力機器制御装置に**機器オーダ**(device order)というビット列を送り出す．入出力制御装置はそれを解読し，それに従った機能を機器にさせる．

チャネルコマンドを実行すると，一つ以上の機器オーダを入出力制御装置に送り出すことになる．機器に依存したオーダの大部分は，チャネルの制御コマンドによって入出力制御装置に送られる．たとえば，次のようなオーダがある．

テープの巻き戻しとテープの逆進(backspace),
テープカートリッジの装填,
アクセス機構の位置決め．

データチャネルは，自分だけで動けるものであるから，複数個のデータチャネルのついている CPU は，多数の入出力操作を保持できる．CPU が持てるチャネルの数は，CPU の記憶サイクルとデータチャネルのデータ転送率とに

よって決まる．

　m は，CPU の記憶サイクル，
　d_i は，i 番目のデータチャネルのデータ転送率

としよう．

そこで，データチャネルの数 n は，次のように決まる．

$$\frac{1}{m} \geq \sum_{i=0}^{n} d_i.$$

たとえば，記憶サイクル 1.5μsec/バイトをもつ CPU は，データ転送率が 275 キロバイト/秒(kb/sec)のデータチャネルを2つもつことができる．何故ならば，

$$\frac{1}{m} = \frac{1}{1.5} = .666 \quad \text{バイト}/\mu\text{sec}$$

$$= 666 \quad \text{kb/sec} \tag{9}$$

$$d_1 = 275 \quad \text{kb/sec}$$

$$d_2 = 275 \quad \text{kb/sec}$$

$$\sum_{i=1}^{2} d_i = 550 \quad \text{kb/sec}. \tag{10}$$

(9)と(10)とを比較すると，

$$\frac{1}{m} > \sum_{i=1}^{2} di$$

が成り立つ．

直観的には，上の数字に従えば，CPU は，チャネルにできるだけ多くのサイクルを "取らせて"，並行的な複数の入出力操作を保持して両方のチャネルが働き続けるようにできる．したがって，複数個のデータチャネルをもつ目的は，CPU が多数の入出力操作を保持できるようにすることである．

2.3.4　釣り合いのとれたチャネルと機器の使用

オペレーティング・システムの入出力機器は，その型と，つながっているチャネルとによってグループに分けることができる．各型ごとに，記号名を割り当ててもよい．ユーザのプログラムがこの記号名を用いて入出力機器の使用を要求したときには，オペレーティング・システム――IOCS あるいはリンキングローダ――は，この割り当てに対する物理機器を選び出す．選択のアルゴリ

ズムは，次の基準に基づいている．

1. その機器が，より多くの使用可能な機器のついているデータチャネルにつながっている．
2. その機器が，準備のできている状態，すなわち，初期設定を必要としていないものである．
3. その機器は，他のものに比べて入出力操作の要求が少なかった．

上述の選択基準によれば，テープやディスクパックの装填をするために機械を停止する回数を最小にし，また，チャネル間の入出力操作を効率良く重ね合わせて釣り合いのとれたチャネルの使用ができる．記号による機器割り当てを使うためには，システムの入出力機構の特性を操作可能な情報にすることが必要であるが，これについては後節で述べる．割り当てアルゴリズムの実行と，割り当てた機器とプログラムとの間の情報の“径路”を確立するためには，**入出力制御システム**(IOCS——Input and Output Control System)というシステムプログラムが必要になる．使用可能な入出力機構をプログラムで有効に使うために，言語翻訳系——たとえばアセンブラ——は，通常，物理機器の割り当てをロード時まで遅らせている．したがって，リンキングローダがIOCSの助けを借りて，実際の割り当てを行なう．

2.4 入力，出力および計算操作の重ね合わせにおけるバッファの割り当てとチャネル多重度の影響

バッファの割り当てとチャネルの利用の議論からわかることは，計算機の記憶サイクルが充分短かく記憶の大きさがかなり大きければ，計算機は多数のチャネルを使うことにより入力および出力操作の多重的並行処理を行なえるだけでなく，プログラムを計算限界または入出力限界にするためにバッファの大きさを調整して，プログラムの動作をある程度制御することもできることである．

多数のチャネルをもつ計算機システムの理想的な状況は，どのようなプログラムの混在に対しても，CPUと全部でなくても一部のチャネルが同時に使用できるようにシステムがプログラムのバッファの大きさを決定できることであろう．すなわち，これらのプログラムによる計算は，チャネルの入力および出力動作と並行的に実行される．この方法では，CPUと全ての必要なチャネルと機器は，いつでも使用されている．CPUといくつかのチャネルは，最大の能力まで同時に使われているので，システムがこれらのプログラムを処理するのに必要な時間は最小になることが推測できる．与えられたプログラムの集合

を最小の時間で処理する計算機システムは，そのプログラムの集合に対して，最大の**処理量**(throughput)を達成したという．

最大の処理量を達成するという目標は，勿論，すべての計算機システムのプログラマが到達したいと思っている目標である．そこに含まれる問題点は，次の通りである．

1. 固定した主記憶においては，バッファ割り当てアルゴリズムは，主記憶の使用可能な空間によって制限を受ける．さらに，大きいと費用がかかる．
2. 与えられたプログラムの集合の入力と出力の動作を事前に知ることは難しいが，これを知らないと，バッファ割り当てアルゴリズムは，うまくプログラムに空間を割り当てることができない．
3. 多数のチャネルは最大処理量を達成するために必ずしも全て必要ではないが，これを維持する費用が多いことを正当化できないであろう．

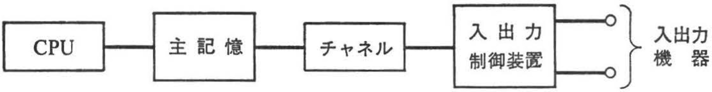
したがって，記憶の大きさ，バッファの割り当てアルゴリズムおよびチャネルの数は，計算機システムの費用/処理量に決定的な影響を持っていると信じられる．

バッファ割り当てアルゴリズムと記憶の大きさの研究は，前の節で説明してきた．ここでは，記憶の大きさとチャネルの多重度について研究しよう．このために，もう一度ある抽象モデルを導入する．これらの抽象モデルは，単純で理想化されているけれども，計算機システムにおける費用/処理量の問題を良く理解する手引きとなる事を望んでいる．

2.4.1　2つのモデルと4つの場合

ここでは，図2.5に示した単一チャネルの計算機システムをもう一度考えよう．また，2つのチャネルをもった計算機システムも考えよう．(図2.6)

ここでは，プログラム P は，3つの操作，すなわち，入力操作 I，計算の集合 C および出力操作 O から成る．これらは同じプログラム P の中にあるので，これら3つの操作は，順次的に実行しなければならない．すなわち，C の開始は，I が終了しない限り出来ない．同様に，O は C が終了したときに初めて開始される．m を，計算機システムが次から次と処理してゆく独立したプログラム $P_1, P_2, \cdots, P_m$ の数とする．入力操作 I_i，計算 C_i および出力操作 O_i をもつ独立したプログラム P_i とは，P_{i-1} の中のすべての操作の終了を待たなくても，P_i のどれかの操作が開始できることを意味している．もっと詳しくいうと



モデルI． 1チャネル計算機システム



モデルII． 2チャネル計算機システム

図 2.6

I_i は，I_{i-1} が終了し次第開始できる．これは，C_i と C_{i-1}，O_i と O_{i-1} にもそれぞれ適用できる．面白い現象として，たとえば，C_{i-1} がまだ続行中であっても，I_{i-1} が終了すれば I_i を開始してもよい．この場合，C_{i-1} は，あきらかに I_i とは独立である．このような現象を，I_i–C_{i-1} **重ね合せ**(overlap)という．簡略化して，これを ***I-C* 重ね合せ**ともいう．一般に，我々は，*I-C*, *O-C*, *I-O-C* の重ね合せの出現に興味がある．

各プログラム P_i は，3つの付随した領域――入力バッファ，作業領域および出力バッファ――をもつ．話を容易にするために，次の領域を一つにまとめる．

1. m 個の全作業領域，
2. m 個の全入力バッファ，
3. m 個の全出力バッファ，
4. プログラム，作業領域あるいはバッファとして使っていない主記憶の使用可能な空間．

一つにまとめた領域を，M とあらわす．

計算機システムのバッファ割り当てアルゴリズムは，4. に示した使用可能な空間を，計算のための追加作業領域あるいは追加バッファとしてプログラムに割り当てることに注意しよう．すなわち，次の事例では，プログラムは，計算限界か入出力限界と考えられるだろう．プログラムが入出力限界であると考えるときには，I_i と O_i とに大きな時間単位を割り当てる．同様に，プログラムが計算限界であるならば，C_i に大きな時間単位を割り当てる．

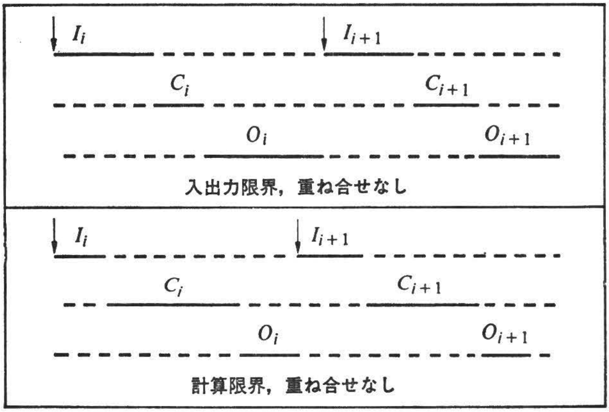


図 2.7　1チャネルによる順次処理

2.4.1.1　事例 I：順次処理

順次処理は，計算機システムを利用する最も簡単な方法である．全領域 M を操作が終るまで専有しているので，追加のチャネルも必要ない．

プログラム P_i を処理するのに必要な時間の総和は，次のようになる．

$$I_i + C_i + O_i.$$

これは，上に示す時系列図に示されているが，そこでは，実線はアクティブな時間をあらわし，破線はアイドルの時間，矢印の対は考えている時間枠を示している．さらに，時間枠は，経過時間をも特徴づけている．

この事例では，操作の重ね合せはない．バッファの切り換えにより一つの領域が入力，出力または計算の操作のために使えるので，全領域 M をこの目的のために使用する．あきらかに，図2.6の単一チャネルの構成で充分である．しかし，この構成においても，重ね合せをする可能性はいくらかある．

2.4.1.2　事例 II：単一チャネルを使った並行処理

チャネルは一つしかないので，I-O 重ね合せはあり得ない．一方，I-C または O-C の重ね合せはあってもよい．そのような重ね合せを達成するためには，領域 M を，入力(又は出力)操作に使えるものと I-C(または O-C)の重ね合せの間計算のために使うものと2つの部分に分割しなければならない．

やはりバッファ切り換えにより，2つの領域のどちらでも，入力，計算，出

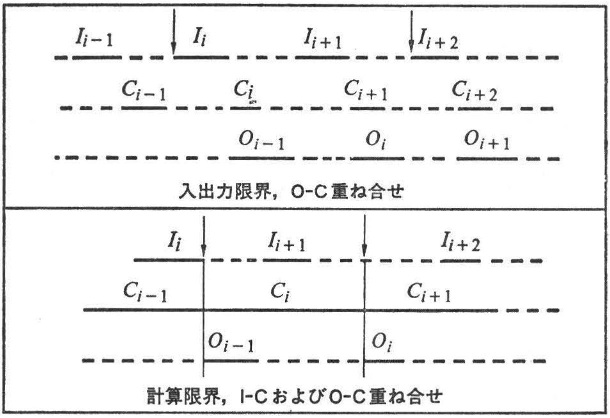


図 2.8　1チャネルによる並行処理

力の操作のために使える．チャネルが多重ではないので，一つの領域で入力操作を続行させ，もう一方の領域で出力操作を続行させることはできない．それゆえ，単一チャネルを採用するならば，*I–C* か *O–C* の重ね合せだけしか達成できない．

さて，入出力限界の場合の時系列を調べてみよう．プログラム P_{i-1} の中の計算 C_{i-1} が終了しても，同じプログラムの中の出力操作 O_{i-1} は，唯一つのチャネルが新しい入力操作 I_i のために使われているので，すぐには始められない．I_i が終了するとすぐに，出力操作 O_{i-1} を始められる．その間，プログラム中の計算 C_i も，使用中のチャネルを要求しないので，実行することができる．したがって，C_i は O_{i-1} と重なる．同じ理由により，時系列図に示したように，C_{i+1} は O_i と重なる．

この場合，プログラム P_i を処理するために必要な時間の総和は，

$$I_i + C_i + O_i$$

となる．しかし，同じ時間枠の中で，P_i が処理されただけでなく，他のプログラムの操作も実行している．これらは，前のプログラムの出力操作 O_{i-1} と，次のプログラムの入力操作 I_{i+1} および計算 C_{i+1} である．前のプログラムと次のプログラムの出力操作がほぼ同じ時間だけかかること，すなわち，$O_{i-1} = O_{i+1}$ が成り立つならば，P_i と P_{i+1} とは同じ時間枠の中で処理されている．

この2つのプログラム P_i と P_{i+1} の経過時間は，

$$I_i + O_{i-1} + I_{i+1} + O_i$$

となる．上式に C_i と C_{i+1} があらわれないのは，それらが O_{i-1} および O_i とそれぞれ重なっているからである．さらに，たとえば P_i のような一つのプログラムの経過時間は，上式の半分と推定する．

t_i をプログラム P_i を処理するための近似的な経過時間とする．このとき，

$$t_i = \begin{cases} \dfrac{1}{2}(I_i + O_{i-1} + I_{i+1} + O_i) \cong I_i + O_i & P_i \text{ が入出力限界のとき，} \\ C_i & P_i \text{ が計算限界のとき，} \end{cases}$$

となる．

2.4.1.3　事例Ⅲ：2つのチャネルを使った並行処理

もう一つチャネルがある場合は，I-C あるいは O-C の重ね合せに加えて，I-O 重ね合せも可能になる．この場合，M の2つの部分を，やはり，入力，出力および計算の操作のために使うことができる．

プログラム P_i を処理するのに必要な経過時間の近似値は，

$$t_i = \begin{cases} \dfrac{1}{4}(I_i + C_i + I_{i+1} + C_{i-1} + O_{i-1} + O_i) & \\ \qquad\qquad \cong \dfrac{1}{2}(I_i + C_i + O_i) & P_i \text{ が入出力限界のとき，} \\ C_i & P_i \text{ が計算限界のとき，} \end{cases}$$

となる．

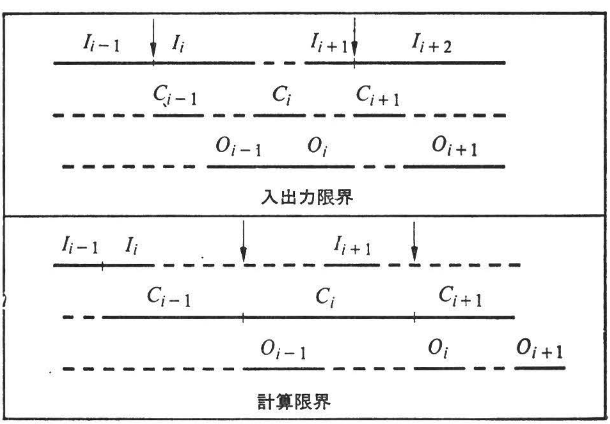


図 2.9　2チャネルによる並行処理

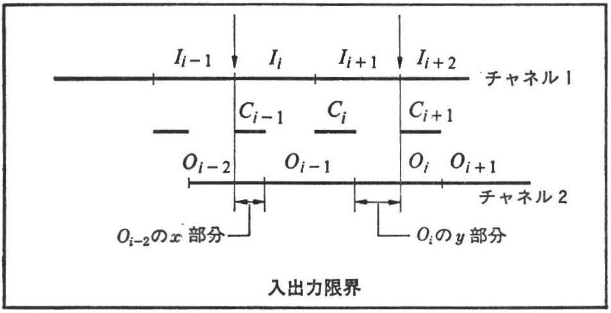


図 2.10　2チャネルによる高度な並行処理

2.4.1.4　事例Ⅳ：2つのチャネルを使った高度な並行処理

最後に，2つのチャネルを使って，I-C-O の重ね合せをどのように成しとげるか考えてみよう．事例Ⅲと同じように入出力限界を考えると，余分の領域があれば，3つの操作を同時に行なうことができる．そこで，M を3つの部分に分割する．計算限界の状況については，ここでは述べない．

プログラム P_i を処理するのに必要な経過時間の近似値は，

$$t_i = \begin{cases} \dfrac{1}{4}(I_i + I_{i+1} + x\cdot O_{i-2} + O_{i-1} + y\cdot O_i) \\ \qquad\qquad\qquad\qquad \cong \dfrac{1}{2}(I_i + O_i) & P_i \text{ が入出力限界のとき}, \\ C_i & P_i \text{ が計算限界のとき}, \end{cases}$$

となる．

2.4.2　処理量の公式

前の節で研究した処理量に対するいくつかの公式を得るために，もう少し簡略化を行ない，各プログラムは，入力，計算，出力の時間の平均値が同じであると仮定しよう．すなわち，各 i に対して，

$$I = I_i, \quad C = C_i, \quad O = O_i$$

が成り立つ．

第4の場合，時間の割合の和は1，すなわち$(x + y) = 1$ と仮定する．さらに，領域 M を2つに分割しなければならない場合には，2つの部分は同じ長さであると仮定する．最後に，記法

$$[a, b]$$

は，a と b のうち大きい方をあらわすとする．たとえば，

$$t = [5,\ 2] = 5$$

である．

プログラム P を処理するための経過時間は，順次処理の場合，

$$T_1 = I + C + O \qquad \text{(重ね合せなし)}.$$

単一チャネルによる並行処理の場合，

$$T_2 = [C, (I + O)] \qquad (I\text{-}C \text{ または } O\text{-}C \text{ の重ね合せ}).$$

2つのチャネルによる並行処理の場合，

$$T_3 = \frac{1}{2}(C + [C, (I + O)]) \qquad (I\text{-}C,\ O\text{-}C,\ I\text{-}O \text{ の重ね合せ}).$$

2つのチャネルによる高度な並行処理の場合，

$$T_4 = \left[C, \frac{1}{2}(I + O)\right] \qquad (I\text{-}C\text{-}O \text{ の重ね合せ}).$$

2.2.1節の式(7)と(8)では，

$$t_{i0} = t_a + nr_{i0},$$
$$t_{CPU} = nr_p$$

が得られていた．

上記の式を用いるときには，(1) バッファ領域は，ちょうど，これら n 個のレコードを収容できる，(2) n レコードはすべて同一の長さである，(3) m 個の各プログラムは，n 個のレコードを処理する，と仮定する．これらの仮定のもとでは，

$$I = O = t_a + nr_{i0},$$
$$C = nr_p$$

が成り立つ．したがって，公式 T_1 は $(t_a + nr_{i0}) + nr_p + (t_a + nr_{i0})$ となり，

$$T_1 = nr_p + 2t_a + 2nr_{i0}$$

となる．

プログラム P は，処理すべきレコードが n 個あり，T_1 は P が n レコードを処理するための経過時間であるから，P が1レコードを処理するのに必要な経過時間 T_1' は，T_1 を n で割って得られる．したがって

$$T_1' = \frac{T_1}{n} = r_p + \frac{2t_a}{n} + 2r_{i0}$$

となる.

P に対するバッファが n レコードを収容できるという仮定に合わせると，まとめた全体の領域 M は，この場合，ちょうど n 個のレコードを収容するのに使われる．すなわち，次の式が得られる.

$$T_1' = \frac{T_1}{n} = \frac{T_1}{M} = r_P + \frac{2t_a}{M} + 2r_{i0}.$$

次に，T_2 を考えよう．代入により，

$$T_2 = [nr_p, ((t_a + nr_{i0}) + (t_a + nr_{i0}))] = [nr_p, (2t_a + 2nr_{i0})],$$
$$T_2' = \frac{T_2}{n} = \left[r_p, \left(\frac{2t_a}{n} + 2r_{i0}\right)\right]$$

が導ける．この場合，まとめた領域 M は，並行処理に対応するために，2つのバッファとして分割してある．各バッファは n 個のレコードを扱えなければならないという仮定に基づけば，$n = (M/2)$ を得る．したがって，

$$T_2' = \frac{T_2}{n} = \frac{T_2}{M/2} = \left[r_p, \left(\frac{2t_a}{M/2} + 2r_{i0}\right)\right] = \left[r_p, \left(\frac{4t_a}{M} + 2r_{i0}\right)\right]$$

が成り立つ.

T_3' および T_4' に対する公式も，同様にして導ける．要約すると，プログラム P が一つのレコードを処理するために必要な経過時間は，4つの場合に対して次のようになる.

$$T_1' = \frac{T_1}{M} \quad = r_p + \frac{2t_a}{M} + 2r_{i0},$$
$$T_2' = \frac{T_2}{M/2} = \left[r_p, \left(\frac{4t_a}{M} + 2r_{i0}\right)\right],$$
$$T_3' = \frac{T_3}{M/2} = \frac{1}{2}\left(r_p + \left[r_p, \frac{4t_a}{M} + 2r_{i0}\right]\right),$$
$$T_4' = \frac{T_3}{M/3} = \left[r_p, \left(\frac{3t_a}{M} + r_{i0}\right)\right].$$

ここで，M は2.4.1節に定義したようにバッファのためにまとめた領域である．これは，場合に応じて，2つの並行処理，あるいは，3つの並行処理を保持するために2つまたは3つの部分に分割される.

計算機システムの**処理量**という概念は，システムがそのプログラムの処理のために提供できる時間の量(すなわち経過時間)によって特徴づけられる．あきらかに，同じ集合のプログラムの処理に少ない時間で済むならば，システムの処理量は高くなる．すなわち，大きな処理量をもつシステムならば，処理を早くできるといえる．上の公式は，プログラム P が一つのレコードを処理するのに必要な経過時間を与えている．これらの経過時間を比較すれば，考えているシステムの相対的な処理量をつかむことができる．

2.4.2.1　計算限界のプログラムに対する効果

ここで，計算限界の処理をする場合を調べてみよう．上の式は，次のようになる．

$$T_1' = r_p + \frac{2t_a}{M} + 2r_{i0},$$

$$T_2' = r_p \qquad r_p > \frac{4t_a}{M} + 2r_{i0} \text{ のとき},$$

$$T_3' = r_p \qquad r_p > \frac{4t_a}{M} + 2r_{i0} \text{ のとき},$$

$$T_4' = r_p \qquad r_p > \frac{3t_a}{M} + r_{i0} \text{ のとき}.$$

これらの式から，次のことが導ける．

1. 順次処理の場合，M を大きくすると入出力装置の長いアクセス時間 t_a を相殺できるので，システムの処理量を改善できる (M が大きいと $2t_a/M$ が減少し，T_1' が小さくなる)．
2. ユーザプログラムを計算限界にするためには，ユーザのプログラムとシステムのバッファ割り当てアルゴリズムを上手に作って，アクセス時間の4ないし3倍以上のものを相殺するような大きな値を M に割り当てるようにする(M が大きくないと，$4t_a/M$ や $3t_a/M$ は入出力装置のアクセス時間 t_a で決まってしまう)．
3. 計算機システムの処理量は，プログラムが大部分計算限界であるならば，並行処理のために多数のチャネルを使っても，大幅な改善はできない．(その処理量は，すべて同じ値 r_p になるからである)．

2.4.2.2　入出力限界のプログラムに対する効果

入出力限界のプログラムに対しては，4つの場合は，次のようになる．

$$T_1' = r_p + \frac{2t_a}{M} + 2r_{i0},$$

$$T_2' = \frac{4t_a}{M} + 2r_{i0},$$

$$T_3' = \frac{1}{2}\left(r_p + \frac{4t_a}{M} + 2r_{i0}\right),$$

$$T_4' = \left(\frac{3t_a}{M} + r_{i0}\right).$$

ここでも，これらの式に基づいて，一般的なことがいくつか言える．

1. 記憶が大きければ，M が大きくなるので，次の式が得られる．

$$T_1' \cong r_p + 2r_{i0},$$
$$T_2' \cong 2r_{i0},$$
$$T_3' \cong \frac{r_p}{2} + r_{i0},$$
$$T_4' \cong r_{i0}.$$

すなわち，2チャネルの計算機システムは，単一チャネルのものよりも2倍の速さでレコードを転送できる．主記憶が大きければ，2チャネルの計算機システムは，単一チャネルのものよりも絶対的によい．

2. 計算の率と転送率がほぼ等しい，すなわち，$r_p = r_{i0}$ が成り立つと，次の式が得られる．

$$T_1' \cong 3r_p,$$
$$T_2' \cong 2r_p,$$
$$T_3' \cong \frac{3}{2}r_p,$$
$$T_4' \cong r_p.$$

すなわち，大きな記憶をもち，計算と入出力転送の率が等しい場合には，*I-O-C* 重ね合せをする2チャネルのシステムは，順次処理の場合に比べて，最大3倍の処理量の改善ができる．

3. 転送率が良くても，主記憶が小さくアクセス時間が遅い場合には，もっと工夫した構成や処理を使っても，順次処理よりもずっと良くなることはない．

2.5　入出力制御システム

入出力操作，プログラムの計算，バッファ，チャネルや機器を扱うのは複雑な仕事なので，ユーザだけでなくオペレーティング・システムも含めた入出力操作の実行を助けるために，**入出力制御システム**(IOCS)が設計された．次節で述べるように，入出力ルーチンは，コーディングの努力を多大に要し，これはユーザプログラムと比較すると釣り合いがとれない．この膨大なコーディングの努力をすることにより，システムプログラマは，アセンブリ時には検出できない誤りや虫取りの苦労を，ユーザから取り除いてくれるのである．

この節では，コーディングの労力の代りに，IOCS の概念的設計に重点を置くことにしよう．

2.5.1　有限状態オートマトンとしての IOCS の設計

この設計は，あきらかに，あらかじめ決めた計算機システムの構成に対して直接行なわれなければならない．この節では，図 2.6 に示した 2 チャネルの構成に対する IOCS の設計の仕方を述べよう．2 チャネルのシステムでは，プログラムを並行的に処理することにより，処理量が大幅に改善できることを知っている．このために，***I-C-O* の重ね合せおよびできれば他の重ね合せもできる IOCS を設計しようと思う．**

IOCS を有限状態オートマトンとして設計するのは，啓示的である．この技法は，恐らく，ソフトウェアの設計の体系的方法として最大の可能性を持っているであろう．

有限状態オートマトンには，状態，状態間の遷移，遷移をひき起す信号がある．この設計では，3 つの領域(すなわち，入力バッファ，計算のための作業領域，出力バッファ)の各々は，任意の時点で 3 つの可能な状態のうちの唯一つの状態になっている．

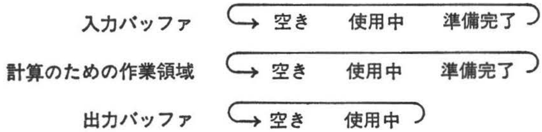
空き————次のデータのブロックに対して使用できる．
使用中———データのブロックを処理している最中．
準備完了——データのブロックの処理を完了して，それを次の領域に転送する準備が出来ている．

“次の領域に転送する” といっても，データのブロックを物理的に一つのバッファ領域から次の領域へ移すということにはならない．その代りに，バッファを切り換えるために，バッファ領域へのポインタの交換を行なう．

遷移をひき起す記号	現在の状態			次の状態		
	入力バッファ	作業領域	出力バッファ	入力バッファ	作業領域	出力バッファ
i	*B*	*I*	*I*	*R*	*I*	*I*
i	*B*	*I*	*B*	*R*	*I*	*B*
i	*B*	*B*	*I*	*R*	*B*	*I*
i	*B*	*B*	*B*	*R*	*B*	*B*
i	*B*	*R*	*I*	*R*	*R*	*I*
i	*B*	*R*	*B*	*R*	*R*	*B*
c	*I*	*B*	*I*	*I*	*R*	*I*
c	*I*	*B*	*B*	*I*	*R*	*B*
c	*B*	*B*	*I*	*B*	*R*	*I*
c	*B*	*B*	*B*	*B*	*R*	*B*
c	*R*	*B*	*I*	*R*	*R*	*I*
c	*R*	*B*	*B*	*R*	*R*	*B*
o	*I*	*I*	*B*	*I*	*I*	*I*
o	*I*	*B*	*B*	*I*	*B*	*I*
o	*I*	*R*	*B*	*I*	*R*	*I*
o	*B*	*I*	*B*	*B*	*I*	*I*
o	*B*	*B*	*B*	*B*	*B*	*I*
o	*B*	*R*	*B*	*B*	*R*	*I*
o	*R*	*I*	*B*	*R*	*I*	*I*
o	*R*	*B*	*B*	*R*	*B*	*I*
o	*R*	*R*	*B*	*R*	*R*	*I*

図 2.11(a) 入力，計算および出力完了の信号によってひき起される状態遷移

これらの3つの領域の状態の遷移は，次のようになる．



(出力バッファは，次の領域をもっていないので，準備完了の状態にはならない．最初の段階では，図2.11(a)および図2.11(b)に示したように，使用中

は B，準備完了はR，空きは I と略記して，状態の遷移を並べて，状態遷移表を作る．また，各状態遷移に対して，その遷移をひき起こす信号の原因も並べる．次に示すような信号が，状態遷移をひき起こすと考えられる．

i　　入力操作の完了，
o　　出力操作の完了，
c　　計算の完了．

しばらくの間，これらの信号によって生ずる状態遷移だけを考えよう．そこで，**図 2.11(a)**の第一行は，作業領域と出力バッファが空いていて(I)，入力バッファが現在使用中(B)のとき，入力操作が完了(i)すると，入力バッファは他の操作に対して準備完了(R)であるという新しい状態に遷移することを示している．信号と状態のあらゆる可能な結合を並べることはできるけれども，あるものは決して起らず，示す必要のないことがすぐにわかる．たとえば，i 信号は入出力操作の完了を示しているので，必ず入力バッファは B でなければならず，したがって，$iIXX$ とか $iRXX$ という形をもった行を並べる必要はない．同様に，$cXRX$，$cXIX$，$oXXI$ というパラメータをもつ行は，排除される．表の各行では，信号から，現在の状態に対応する次の状態が導ける．これは，完了の信号を出した領域の状態だけを R(準備完了) に変えるだけでよい．表を作成するときには，すべての行は，他のすべての行と独立であり，ここでは体系的な数え方になっているけれども，任意の順序で並べることができる．

表の作成の次の仕事は，表の s(IOCS によって発せられる信号)部分である．この主たる目的は，可能なときにいつでも新しい操作を開始することである．この原理を使うと，次の状態のエントリを決定でき，**図 2.11(b)**のように並べることができる．

i, o, c, s の信号のほかに，IOCS はあまり頻繁には出ない 2 つの信号を受け取る．すなわち，プログラムの開始を示す $START$ と，最後のブロックが入力領域に入ったことを知らせる $LAST$ である．これらは，別の行に書くこともできるけれども，表の空間を節約するために，スラッシュで $LAST$ エントリと他のどれかの信号との区別をして書く．

表は，IOCS のプログラムの論理の中心を成す．これは，これから述べる簡単な走査プログラムが参照する．システムは，実行を続けながらある一つの状態にいると考えよう．いつかは，i, o，あるいは c の信号が出されるであろう．前にも示したように，これは割り込みを起し，信号の識別をする．そこで，走

遷移をひき起す記号	現在の状態			次の状態		
	入力バッファ	作業領域	出力バッファ	入力バッファ	作業領域	出力バッファ
START	*I*	*I*	*I*	*B*	*I*	*I*
s/LAST	*I*	*I*	*B*	*B/I*	*I*	*B*
s/LAST	*I*	*B*	*I*	*B/I*	*B*	*I*
s/LAST	*I*	*B*	*B*	*B/I*	*B*	*B*
s/LAST	*I*	*R*	*I*	*B/I*	*I*	*B*
s/LAST	*I*	*R*	*B*	*B/I*	*R*	*B*
s	*B*	*I*	*I*	*B*	*I*	*I*
s	*B*	*I*	*B*	*B*	*I*	*B*
s	*B*	*B*	*I*	*B*	*B*	*I*
s	*B*	*B*	*B*	*B*	*B*	*B*
s	*B*	*R*	*I*	*B*	*I*	*B*
s	*B*	*R*	*B*	*B*	*R*	*B*
s/LAST	*R*	*I*	*I*	*B/I*	*B*	*I*
s/LAST	*R*	*I*	*B*	*B/I*	*B*	*B*
s/LAST	*R*	*B*	*I*	*R*	*B*	*I*
s/LAST	*R*	*B*	*B*	*R*	*B*	*B*
s/LAST	*R*	*R*	*I*	*B/I*	*B*	*B*
s	*R*	*R*	*B*	*R*	*R*	*B*

図 2.11(b) IOCS の介入，プログラムの開始，最後のブロックの入力から生ずる信号でひき起される状態遷移

査プログラムは，次のことを行う．

1. 信号を使って，(図 2.11(c)に示したように)，最初に探索すべき表を識別する．
2. 現在の状態を引数として，この表を探索する．一致が取れれば，次の状態が，新しい現在の状態になる．
3. 新しい現在の状態を，*s* 表への引数として使う．一致が取れれば，次のことを行なう．次の状態が，現在の状態に置き換る．付随した処理が行なわれる．割り込み前の動作または新しい動作に戻って，実行を続ける．一致が取れなければ，設計が誤っていることがわかる．

このIOCSは，いくつかの知的な動作をすれば，処理量を改善することができる．これらの動作は，バッファの切り換え，新しい入力と出力の操作や計算を始めること，新しい操作が開始できないときには，何の行動もしないことを

意味する．新しい操作を始めるときは，IOCSは，これらの操作を重ね合わせることにより，出来るだけ多くの並行処理を達成しようと試みる．したがって，この設計の最初で述べた本来の目的に合致する．

状態遷移表ができると，これを単一の生起信号ごとに4つの小さい表に分割する(最初の3つは図2.11(c)，残りは図2.11(d)参照)．また，ある行の現在の状態が，どこかの次の状態として見つからない(*START*は除く)ときには，それを消去して良い†．*s*表においては，各状態遷移は，IOCSからの信号によって

i 表

状態	
現在	その次
BII	*RII*
BIB	*RIB*
BBI	*RBI*
BBB	*RBB*
BRI	*RRI*
BRB	*RRB*

c 表

状態	
現在	その次
IBI	*IRI*
IBB	*IRB*
BBI	*BRI*
BBB	*BRB*
RBI	*RRI*
RBB	*RRB*

o 表

状態	
現在	その次
IIB	*III* → A USER
IBB	*IBI* PROGRAM
IRB	*IRI* ENDS
BIB	*BII*
BBB	*BBI*
BRB	*BRI*
RIB	*RII*
RBB	*RBI*
RRB	*RRI*

図 2.11(c)　IOCSの介入を必要としない遷移—1

† IOCSの設計者は，全ての可能な状態を調べるために並べつくしてもよい．実際に，生起信号の各型ごとに324(= (3 × 3 × 2)²) 個の状態を調べなければならない．幸いなことに，多くの遷移は生起しないので表から消去できる．

sおよびs/LAST 表

信 号	状 態 現 在	その次	行なわれる操作．前の動作に戻る．	達成できる重ね合せ
s/START	*III*	*BII*	入力(およびプログラム)を開始する．	
s/LAST	*IIB*	*B/I, IB*	入力を開始する．*LAST*（操作なし）	*I-O/*
s/LAST	*IBI*	*B/I, BI*	入力を開始する．*LAST*（操作なし）	*I-C/*
s/LAST	*IBB*	*B/I, BB*	入力を開始する．*LAST*（操作なし）	*I-C-O/C-O*
s/LAST	*IRI*	*B/I, IB*	**c**から**o**へ移る．（すなわちバッファを切り換える）．出力と入力を開始する．	*I-O/*
s/LAST	*IRB*	*B/I, RB*	入力を開始する．*LAST*（操作なし）	*I-O/*
s	*BII*	*BII*		
s	*BIB*	*BIB*		
s	*BBI*	*BBI*		
s	*BBB*	*BBB*		
s	*BRI*	*BIB*	**c**から**o**へ移る．（すなわち，バッファを切り換える)出力を開始する．	*I-O*
s	*BRB*	*BRB*		
s/LAST	*RII*	*B/I, BI*	*i*から**c**へ移る．（すなわち，バッファを切換える．）計算を開始する．*LAST*でなければ入力を開始する．	*I-C/*
s/LAST	*RIB*	*B/I, BB*	*i*から**c**へ移る．（すなわち，バッファを切り換える．)計算を開始する．*LAST*でなければ入力を開始する．	*I-C-O/C-O*
s.	*RBI*	*RBI*		
s	*RBB*	*RBB*		
s/LAST	*RRI*	*B/I, BB*	*c*から*o*, *i*から*c*へ移る．（すなわち，バッファを切り換える．）出力を開始する．計算を開始する．*LAST*でなければ，入力を開始する．	*I-C-O/C-O*
s.	*RRB*	*RRB*		

図 2.11(d) IOCS の介入を必要としない遷移—2

生ずる．その信号は，システムの処理量を改善するように，新しいプログラムを開始するか，出来るだけ多くの重ね合せをするために使われる．*LAST* の信号によっても，IOCS は動作を起こすから，*s*/*LAST* 表は，IOCS が本当に知的に動作するということを示している．知的に動作するとは，バッファの切り換え，操作の開始，重ね合せをすること，したがって，途中の状態をスキップするような能力を意味する．

たとえば，入力完了の信号 *i* が生じて計算機の状態が *BIB* から *RIB* に変る(図2.11(c)の *i* 表参照)ときに，IOCS が行なう上述の **3.** の動作を，詳しく考えてみよう．並行処理を利用すればシステムの処理量が改善されることを知っているので，IOCS は次の動作をする．

a) 入力バッファと作業領域との切り換え(入力バッファの中のデータは計算のために準備完了(*RIB* の中の *R*)となっており，作業領域は空き(*RIB* の中の *I*)であるから，バッファの切り換えは，物理的にでなく実効的にデータを入力バッファから作業領域へ移す)．

b) 作業領域の中での計算の開始(ここまでで，計算機の状態は *IBB* となっている．新しい計算によりシステムの処理量は大幅に改善されているけれども，IOCS は，もっと工夫できる．しかし，入力バッファは空いていることに注意しよう)．

c) 入力バッファが空き(*IBB* の中の *I*)なので，そのバッファに対する新しい入力操作を開始する．計算機の状態は，最終的には *BBB* となり，*I-C-O* の重なりにより高度な並行処理をしている．並行処理の程度に制限を加えてしまう唯一つの特別な状況は，新しい入力操作がないとき，すなわち，最後のレコードが前の入力操作で読まれてしまったときである．この場合，状態は *IBB* となり *C-O* の重ね合せだけをしている．この例は，図2.11(d)の一つのエントリとして示してある．

2.5.2 処理系とサブシステムの層としての IOCS の設計

IOCS を有限状態オートマトンとして設計する方法は啓示的ではあるけれども，読者に IOCS の設計を単純視させるきらいがある．設計の範囲を理解するために，設計を難しくするいくつかの要因を考えてみよう．

A. ユーザの要求

ユーザの立場から見れば，入出力命令を出すことは，他の命令を出すことと大きく違っているわけではない．すなわち，ユーザは，次のことを期待する．

1. 入出力命令は，単純な記号で書く．
2. 二次記憶領域への参照は，記号名と相対番地とで行なう．
3. 二次記憶機器へ格納したり，そこから検索してくるデータは，機器の物理的な特性を無理に考えることなく，ユーザが望んだもとの形式でなければならない．
4. システムは，入出力操作を効率良く行なう．

B. システムの設計基準

ユーザの要求に合わせるために，IOCS は次のことができなければならない．

1. 入出力命令，それに関係するチャネルコマンドと機器オーダの正しい形成に，ユーザが関心を払わないでも済むようにする．
2. 二次記憶領域に記号名をつける．
3. ユーザのデータを格納してある機器を保護する．
4. ブロッキング，デブロッキング，および，バッファの割り当てを行なう．
5. システムの処理量を改善する．

上の基準は，IOCS が，機器の特性や利用法，記号名をつけた記憶装置の大きさと使用状況，機器と(制御)装置との関係などに関する情報を持つことを要請している．この情報により，IOCS は次の 4 つの機能を果すことができる．

1. ユーザの入出力要求の解釈，
2. 必要な入出力操作の実行，
3. データや機器の検証，
4. ユーザのデータのバッファリング．

2.5.2.1 入出力操作，チャネルおよび機器に関する情報の構成

IOCS は，ユーザに対し入出力ルーチンを提供し入出力操作を実行するので，システム内の使用可能な入出力機構について知っていなければならない．このために，これらの機構に関する情報を表にしておき，システム生成時にこれを最初に初期設定し，以後，システムを操作している間ずっと IOCS がこれを保守する．この情報を構成し，IOCS を作り出す唯一ではないが一つの方法を示そう．

2.5.2.1.1 物理機器

入出力機器は，それぞれ，**機器制御ブロック**(device control block)と呼ぶ

表をもっている．機器制御ブロックには，入出力機器の現在の状態と，機器を初期設定したり保守したりできるルーチンのエントリに関するすべての情報が入っている．次に示すような情報が含まれている．

(D1)　機器が使用中かどうか，機器のつながっているチャネルが使用中かどうか，機器が使用の要求を受けているかどうかを示すフラッグ，
(D2)　この機器から読み出したか，または，書き込んだ物理レコードの数，
(D3)　この機器の入出力操作に対する命令，チャネルコマンド，機器オーダの適正な順序を作り出せるルーチンの番地，
(D4)　操作の進行を追跡できるルーチンの番地，
(D5)　誤りを処理し回復できるルーチンの番地，
(D6)　入出力操作中に生じた読み/書きの誤りで，永久的なものおよび回復したものに関する統計，
(D7)　同一のチャネルにつながっている次の機器の制御ブロックへのポインタ，
(D8)　記号装置名が機器に割り当てられている場合，記号装置名の表へのポインタ(あとで説明する).

上記のことから，ユーザは，入出力操作に対して命令，チャネルコマンド，機器オーダの正しい列を書く必要がないことがわかる．これらの動作は，機器制御ブロックに番地が入っているルーチンによって行なわれるからである．さらに，入出力操作の進行のチェックと誤りの回復とは，それらのルーチンとそれらに関する統計とをIOCSが機器制御ブロックの中にもっているので，システムの関数として行なわれる．

(D7)からわかるように，同一のデータチャネルにつながっている機器のすべての機器制御ブロックは，ポインタで結合してある．それゆえ，IOCSは，一つの機器制御ブロックから次のものへとポインタをたどることにより，そのチャネルに対する機器の数やそれらの動作を監視することができる．さらに，チャネルの使用に対する機器の優先度をつけるためには，最高の優先度をもった機器制御ブロック(なんらかの指示子でマークしてある)が，次の優先度をもつものを指すようにそのチャネルの機器制御ブロックを並べておくだけでよい．

2.5.2.1.2　記号装置

一般に，ユーザは，入出力機器を記号名で指定する．記号名を使用すると，

多くの利点が得られる.

1. 記号名は使い易く，名前のあらわす機器の型をあらわすように意味づけることができる.
2. 機器を記号装置で参照し，システムが記号装置と物理機器との対応をつけるようにしておけば，一方で，機器やチャネルの使用の釣り合いをシステムが取れるようにでき，また一方で，ユーザがそのような配慮をしないで済むことになる.
3. 記号装置名が名前である限りは，記号装置と物理入出力機器との実際の対応づけを，時間的に変えることができる．したがって，一度書かれてコンパイル(あるいはアセンブル)されたユーザのプログラムは，物理機器の構成がたとえ変更されてしまっても，あとで実行することができる.

記号装置は，常に一つの物理装置と対応するわけではない．ディスクやドラムのような機器は，通常，多くの部分に分割され，これらの機器は**直接**アクセスできる．すなわち，アクセス機構を直接分割した部分に動かして位置を見つけられる能力をもっているので，それぞれの部分に記号装置を割り当てる．したがって，一般に，記号装置と直接アクセス機器との間には，多対一の対応がつけられる．テープのある部分に到達するためには，バッファを作らなければならない．指定した情報が見つかるまで，テープ上のレコードを順次に読み込まねばならず，時間と空間を消費する処理である．このために，記号装置と順アクセス型入出力機器とは，一対一の対応がつけられる.

各記号装置に対して，**装置制御ブロック**(unit control block)を作る．物理機器を制御するための情報やルーチンを提供する機器制御ブロックと同様に，装置制御ブロックは，記号装置の情報をもっている．複数個の記号装置が一つの物理装置を共有するかもしれない(多対一の対応がある)ので，同一の機器に対して一つ以上の入出力操作が見られるかもしれない．個々の入出力操作に関する情報は，装置制御ブロックの中にある．したがって，装置制御ブロックは，個々の入出力操作の進行を監視する情報を提供する.

装置制御ブロックがもつ基本的な情報は，次の通りである.

(U1) 機器の型と使うチャネル,

(U2) この装置に現在要求されている入出力操作,

(U3) 機器の入出力操作に対する命令，チャネルコマンド，機器オーダの適正な列を作り出せるルーチンの番地,

(U4) 操作の進行を監視できるルーチンの番地,
(U5) 誤りを処理し回復できるルーチンの番地,
(U6) この装置に割り当てられた直接アクセス機器の分割部分の先頭と終端の番地,
(U7) 直接アクセス機器の現在の番地，番地当りのレコードの数，および，現在のレコードの数に関する統計,
(U8) そこに割り当ててある記号装置名表内の記号装置名へのポインタ(次節で説明).

(U3), (U4), (U5)における番地を，装置制御ブロックの実際のルーチンと別のものに置き換えれば，IOCSは，ユーザに自分自身のルーチンを使うという選択を提供することができる．この選択を実施する場合には，IOCSは，それが用意した標準の番地のかわりに，装置制御ブロックにユーザのルーチンの番地を単に代入するだけでよい．

2.5.2.1.3　記号装置名

記号名と入出力機器との対応をつけるために，**記号装置名表**(symbolic unit name table)を構成する．表の各エントリは，次の3つの部分から作られている(図2.12).

(S1) 記号名,
(S2) この記号名があらわす記号装置の装置制御ブロックへのポインタ,
(S3) この記号装置が割り当てられている入出力機器の機器制御ブロックへのポインタ.

2.5.2.2　IOCSの4つの基本機能

入出力制御システムは，4つの基本サブシステム，すなわち，入出力操作の解釈と実行をすること，これらの入出力操作によって処理されるデータの検証とバッファリングを行なうことから構成される．それらは，入出力操作解釈系，入出力操作実行系，入出力検証サブシステムおよび入出力バッファリング・サブシステムという．

2.5.2.2.1　入出力操作の解釈

入出力操作に対する入出力命令，チャネルコマンドおよび機器オーダは，機器に依存したものなので，注意深く面倒な準備を必要とする．ユーザとシステ

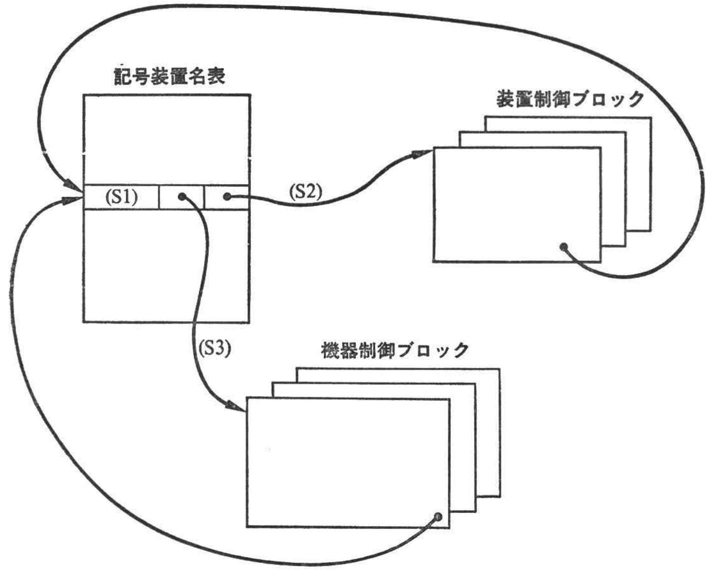


図 2.12 記号装置名表

ムプログラマからこの準備の苦労を取り除くために，**入出力操作解釈系** (I/O operation interpreter)を用意して，入出力操作に対する入出力命令，チャネルコマンド，機器オーダの適正な列を解析し生成するために使う．入出力操作は，一般に，次のものから構成される．

1. 入出力操作の名前，
2. 関与する記号装置の名前，
3. 操作に必要なパラメータのリスト，
4. 正常および誤りの場合の戻り場所のエントリ．

入出力操作の例として，いくつかの機器における読み/書き操作がある．

入出力操作に関する情報は，(S2) で指定した記号装置の装置制御ブロックの中に記録してある．

2.5.2.2.2 入出力操作の実行

命令，コマンド，オーダの適正な列が実行可能となったとき，実行を遂行する**入出力操作実行系**(I/O operation executer)にこの列を渡す．

入出力操作実行系は，データチャネルの使用のスケジュールをし，チャネル

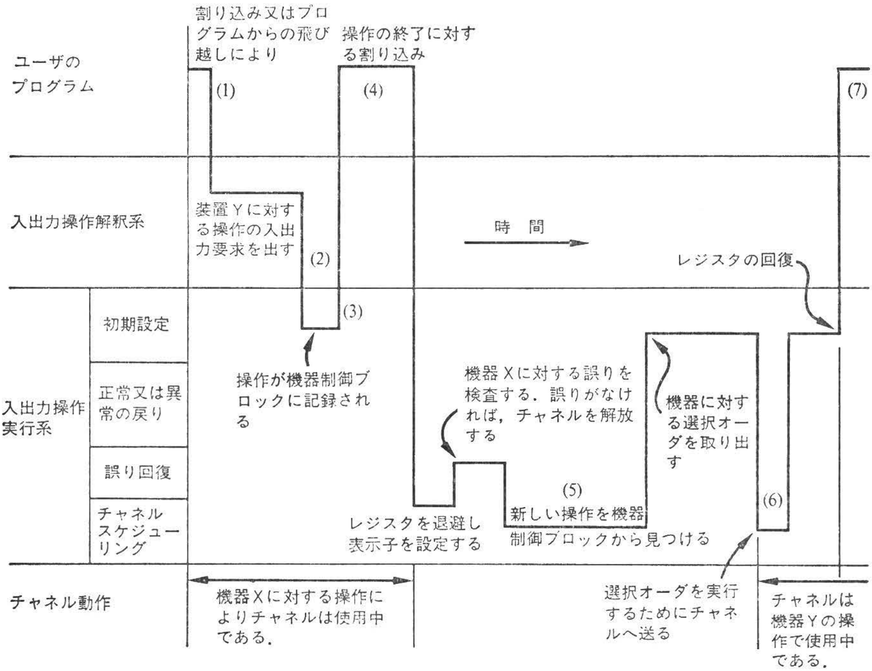


図 2.13　入出力操作解釈系と実行系の役割

の実行を監視し，誤りを回復し，コード変換をし，正常および誤りの戻りを行なう．チャネルの動作，誤り回復ルーチンの場所，戻り番地に関する情報は，機器制御ブロックがもっており，入出力操作実行系が管理をする．システムの処理量を改善するための操作の重ね合せは，このレベルで行なう．

図 2.13 に，入出力操作の解釈系と実行系の役割を，動作の筋書きと一緒に説明してある．

ユーザプログラムの入出力の要求は，通常，システムマクロの形をとっている(演習問題7参照)．あきらかに，マクロ機能をもったアセンブラが，このマクロを機械命令とパラメータのリストの列に展開する．その一つの機械命令は，スーパーバイザコール(SVC)様のもので，これによりユーザのプログラムは解釈され，実行中に IOCS に飛んでゆく．そのような制御の移動を，**割り込み**という．一方，スーパーバイザコールをもっていない計算機では，制御の移動は，標準的なプログラムの飛び越し命令を使って行なう．割り込みにしろプログラム飛び越しにしろ，この命令により，ユーザの入出力要求は入出力操作

解釈系に伝えられる． これは， 図2.13のステップ(1)に示してある． 解釈系は，次に，どの機器でこの入出力要求を処理するのかを決める．前にも述べたように，装置に対する機器の割り当ては，時間と共に変ってもよかった．この情報は，マクロをアセンブルする時には使えない．しかし，装置と機器の制御ブロックおよび記号装置名表の中で特定の機器を考えている装置に最後に割り当てるのは，IOCSが決定し，それを保守する．マクロ中のパラメータのリストと制御ブロックとを参考にすれば，機器を決定できる(2)．

実行系は， 機器Yがつながっているチャネルの使用をスケジュールする．この例のようにチャネルが使用中であれば，そのチャネルの使用は遅らせなければならない(3)．一つの選択は，チャネルを待っている時間を，プログラムの他の部分を実行するために使えばよいと考えて，ユーザのプログラムに戻ることである(4)．チャネルが空き次第，実行系は，機器Yに対してチャネルをスケジュールする(5)．同一のチャネルにつながっている機器の制御ブロックは，ポインタにより結合されていることを思い出そう． そこで．機器Xの機器制御ブロックを介して，チャネルの待ち行列に入る次の機器制御ブロックを決めることができる．この場合，次は機器Yの機器制御ブロックである．続いて，機器を働かせる(6)．ここまでくれば，チャネルと機器とは，読み込みや書き出しのような入出力操作を実行する準備ができる．そこで，ユーザプログラムに戻って，操作を続ける．

2.5.2.2.3 データとデータ記憶の検証

入出力操作の実行系は， 事前に決められた入出力操作を実行するだけなので，その入出力操作によって処理しているデータが正しいものかどうかは，わからない．誤ったデータに対する入出力操作の開始を防ぐために，データをファイル中のブロックとして構成してある．ラベルを作り，ファイルに割り当てることができる．ファイルを使うたびにラベルを検証すれば，ファイルは適切な機器の上にある正しいファイルであることをシステムは確かめることができる．**入出力検証サブシステム**(I/O verification subsystem)は，ラベルの検証と作成の機能をすべて実行する．ユーザは，このサブシステムにラベルの名前を与え，それは媒体上に標準的な形式でラベルやレコードを格納する．たとえば，テープを記憶媒体として使い，ファイルは2つのテープを占めるほどの長さであったとすると，サブシステムは，最初の巻の末尾にファイル終端継続ラベルを作り，巻を切り換え，最初のテープを“閉じ”，第2のテープを“開き”，第2のものに先頭ラベルを書き出し，最後に，第2のテープにファイル終端継

続ラベルをつけてそれを閉じる．レコードギャップを作ったり，見つけたりするのも，入出力検証サブシステムである．これらのギャップは，テープ上のレコードやラベルを分離するために使う．

2.5.2.2.4　データのバッファリング

外部機器上の記憶空間を最適に使い，またそれに最短にアクセスするために，IOCSは，ユーザの論理レコードを記憶の物理レコードとして構築する機能を果す．さらに，物理レコードが機器から取り出されたとき，IOCSは，それをユーザプログラムに論理レコードとして戻す機能も行なう．物理レコードの大きさの種類は，物理機器の型の種類と同じ位あるから，このような機能を行なうIOCSの仕事はかなり大変である．しかし，ユーザは，自分の論理レコードとシステムの物理レコードとの対応関係は知らない．

論理レコードを物理レコードに構成する過程は，前に述べたように，**ブロッキング**という．IOCSは，物理レコードよりも少し大きめのバッファに，論理レコードを一時に一個づつ置いて，ブロッキングを行なう．(バッファは，2，3の制御語の空間も含んでいる)．バッファが一杯になれば，その内容を物理レコードまたはブロックとして，機器上に書き出す．取り出した物理レコードを論理レコードに分離し，一時に1個づつユーザのプログラムに与える過程を，**デブロッキング**という．ブロッキングおよびデブロッキングに加えて，入出力操作と計算の重ね合せが効果的になるために入出力操作実行系がバッファの切り換え操作を行なう必要がある．上記の機能を実行するIOCSの部分を，**入出力バッファリング・サブシステム**(I/O buffering subsystem)――IOBS――という．

前に述べたように，リンキングローダのバッファ割り当てアルゴリズムは一つの入出力操作に複数個のバッファを割り当ててもよかった．IOBSの役割は，これらのバッファを利用して，読み込みや書き出しによる連続的な情報の流れが維持されるようにすることである．バッファが一杯(空)であるときは，IOBSは，すぐにデータの流れを別のバッファへ(別のバッファから)動かす．データが別のバッファへ(別のバッファから)流れている間に，IOBSはこのバッファのデブロッキング(ブロッキング)を行なう．こうして，計算と入出力操作の重ね合せができる．

2.5.2.3　物理的および論理的入出力制御システム

IOCSの一番下のレベルは，入出力操作実行系であることに注意しよう．入

出力操作に対する入出力命令，チャネルプログラム，機器オーダは，適正な順番と形式をしていると仮定しているので，命令はCPUで処理でき，チャネルコマンドはデータチャネルに転送でき，機器のオーダは，適当な応答を期待して機器に転送できる．その上のレベルには，入出力操作を識別し，その操作のために入出力命令，チャネルコマンド，機器オーダの適当な列を作り出す入出力操作解釈系がある．IBM OS/360のようなあるオペレーティング・システムでは，個々の操作に対する命令，コマンドやオーダの列は，すでに組み立てられ主記憶に格納されている．入出力解釈系は，単に操作を識別し，それに対応する列を取ってきて，必要ならばその列をデータチャネルに転送する．さらに高いレベルには，ラベルを読んだり書いたりする操作をする入出力検証サブシステムがある．この場合，解釈系と実行系は，これらの操作を実行するために利用される．一番高いレベルには，入出力バッファリング・サブシステムがある．あるファイルのラベルが正しいと検証できなかったり作成できなかったりするならば，そのファイルに対する残りの入出力操作を実行する必要はない．ラベルが正しく一致したり作成されたりしたときには，入出力バッファリング・サブシステムが，残りの入出力操作の実行をする．これら4つの入出力サブシステムと処理系の集まりを，**入出力制御システム**という(図2.14)．

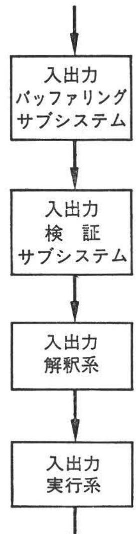


図 2.14 入出力制御システム

入出力プログラミングの機構を層構造にするときには，IOCSはユーザやシステムプログラマの要求に応じて，サブシステムを少くしたり多くしたりすることができる．たとえば，ユーザが自分のファイルに標準でないラベルを書きたいとき，解釈系と実行系だけを目的のために使って，入出力ラベル検証サブシステムを回避することができる．入出力バッファリング・サブシステムが提供しているバッファリングの機構が不適当であるとユーザが判断するならば，もっと工夫をこらしたバッファリング方式を採用することもできる．そこで，ユーザは，入出力バッファリング・サブシステムを回避し，自分自身のバッファリング・システムを導入できる．

一方，IOCSは，ユーザに次のことを簡単に行なう方法や規則を提供できる．

1. 入出力操作を要求の形式（たとえば，マクロ）で指定して，入出力命令，チャネルコマンド，機器オーダの正確な形式にわずらわされないようにすること，
2. ファイルのラベルを作成したり検証したりすること，
3. 論理レコードを読んだり書いたりすること．

それと同時に，ユーザには見えないけれども，IOCSは，次のことをして物理的な機構をもっと有効に利用しようと努めている．

1. データを物理レコードの形で格納し探索すること，
2. 機器やチャネルの操作の終了を検出し，別のものを開始する能力を提供すること，
3. バッファの切り換えにより，連続的なデータの流れを維持すること，
4. 計算と入出力操作の重ね合せを許すこと．

ここで論じているIOCSは，2.5.1節に述べたものよりも，ずっと精密なものである．両者とも，物理レコードの格納と探索ができ，機器やチャネルの操作の終了を検出でき，バッファ切り換えによるデータの流れを維持でき，入出力と計算の重ね合せをすることができるけれども，そこには少々の違いがある．2.5.1節で述べたIOCSは，論理レコードを扱えず，ラベルを検証できず，デブロッキングやブロッキングの操作をしていない．そこでは，物理レコードと物理入出力操作を扱うように制限していたので，このIOCSを**物理**IOCS(PIOCS)と名づける．同じ理由で，この節で述べたIOCSは，しばしば，**論理**IOCS(LIOCS)といわれる．

2.6 あ と が き

最初に述べたオペレーティング・システムを有限状態オートマトンとして設計することは，[6] で試みられた．この方法を IOCS の設計にあてはめて説明するために，[4] にある良く作られた例題を借りてきた．処理量の公式の議論は，[5] にあるものを少し簡単化して使っている．バッファ割り当てアルゴリズムは，[3] による．また，章末に示した IOCS の一般化モデルは，[7]，[8]，[9] に基づいている．

オペレーティング・システムを，その副機能や機能の "層" を使って構成するという概念は，[1] に帰するものであるが，[7] に概略が述べてあるような IOCS を処理系とサブシステムのレベルと考えることは，この概念の先駆者と見做してもよいかもしれない．

参 考 文 献

1. E. W. Dijkstra. "The Structure of the THE System." *Comm. of the ACM* **11,** 5 (May 1968): 341–346.
2. J. J. Donovan. *Systems Programming.* New York: McGraw-Hill, 1972, pp. 350–366 and 401–424.
3. R. Hedberg. "Design of An Integrated Programming and Operating System—Part III: The Expanded Function of the Loader." *IBM Systems Journal* **2** (September-December 1963): 298–310.
4. H. Hellerman. *Digital Computer System Principles.* New York: McGraw-Hill, 1967, pp. 115–135.
5. H. Hellerman and H. J. Smith, Jr. "Throughput Analysis of Some Idealized Input, Output, and Compute Overlap Configurations." *Computing Surveys* **2,** 2 (June 1970): 111–118.
6. R. E. Heistand. "An Execution System Implemented as a Finite State Automaton." *Comm. of the ACM* **7,** 11 (November 1964).
7. "IBM 7040/7044 Operating System (16/32K) Input/Output Control System." File No. 7040-30, Form C28-6309-5, IBM Corp., (September 1966).
8. G. H. Mealy. "Operating Systems." Rand Report No. P-2584, The Rand Corp., Santa Monica, California, May 1962, AD 603146.
9. A. Padegs. "The Structures of System/360—Part IV: Channel Design Considerations." *IBM Systems Journal* **3,** 2 (1964): 165–180.
10. C. P. Wang and W. Anacker. "Performance Analysis of Multiprocessor Systems with Hierarchical Memories." *IEEE Trans.* **EC-16** (December 1967): 764–773.
11. P. Wegner. *Programming Languages, Information Structures, and Machine Organization.* New York: McGraw-Hill, 1968, pp. 66–72.
12. P. White. "Relative Effects of Central Processor Input-Output Speeds Upon Throughput on the Large Computer." *Comm. of the ACM* **7,** 12 (December 1964): 711–714.

演 習 問 題

1. 小さなバッファを用いて入出力操作を行なう場合に特別の配慮をしたバッファ割り当てアルゴリズムを提案せよ．そのアルゴリズムを，例を使って説明せよ．

2. バッファの大きさに関係なく，ある特定の型の入出力操作に特別の配慮をしたバッファ割り当てアルゴリズムを提案せよ．そのアルゴリズムを，例を使って示せ．

3. (Hellerman と Smith) かっこの中の英字が次の構成をあらわしているグラフ(図2.15)を調べよ．

a ——順次型，すなわち重ね合せなし．

b ——入力と計算または出力と計算のどちらかが，同時に行なわれてもよい．

c ——入力，出力および計算操作の3つの組み合せのうち2つが同時に行なわれてもよい．

d ——入力，出力および計算操作が，同時に行なわれてもよい．

たとえば，(b < d < a < c)は，処理量からみた相対的性能が，構成 b は d より悪く，それは今度は a より悪く，さらにそれは c より悪いことを示

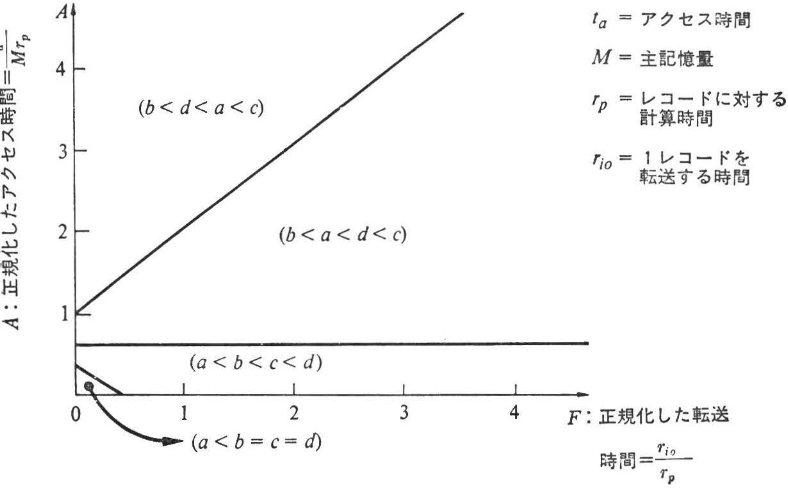


図 2.15

している．

これを調べた後で，グラフの解釈をせよ．参考のために，プログラム P が一つのレコードを処理するために必要な経過時間は，4つの構成に対してここにもう一度示しておく．

a ——$r_p + (2t_a/M) + 2r_{i0}$,
b ——$[r_p, (4t_a/M + 2r_{i0})]$,
c ——$1/2(r_p + [r_p, 4t_a/M + 2r_{i0}])$,
d ——$[r_p, (3t_a/M + r_{i0})]$.

4. 3つのチャネルをもつ計算機システムに対する処理量の公式の集合を求めよ．

5. (F. N. Ng)2.5.1節のPIOCSは，状態遷移図によっても記述することができる．この図は，計算領域と入出力バッファの状態に基づいて，遷移の列を生成する．I, C, O, S の状態のほかに，先読み状態 LO と LC とを導入する．LO のときは，状態 C の状況を調べる．それが準備完了であれば，計算領域と出力バッファを切り換える．さらに，C の状態を I に，O の状態を B に変更し，次の状態を C とする．状態が LC のときも，同様の手順の動作をする．遷移図を，ここに示しておく．(図2.16).

2つの先読み状態を使わないで，等価な遷移図を作成せよ．

6. 図2.11(d)の遷移表で示したPIOCSを実現せよ．PIOCSの仕事を説明するために，図2.11(a)や図2.11(b)に示した状態遷移の生起がPIOCSの入力として使えるように，システムの環境を設計しそれを実現せよ．

7. 2.5.2節で概略を示したLIOCSを使用するための入出力マクロの集合を設計し，その流れ図を示せ．これらのマクロは，次のような論理的操作を含んでいなければならない．

ファイルを開くこと．
ファイルを閉じること．
論理レコードの検索．
論理レコードの格納．

上記の操作を使うためには，ユーザは，自分のファイルとレコードに関するいくつかの情報を与えなければならない．さらに，LIOCSは，装置および機器制御ブロックと記号名表とから，ブロック，バッファ，ファイルとレコードについての機器特有のパラメータに関する情報を引き出さねばならな

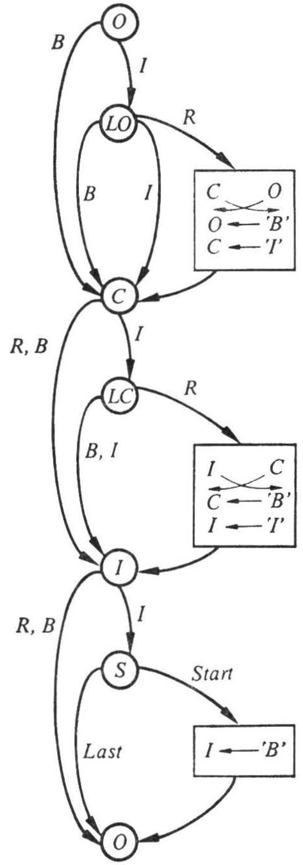


図 2.16

い．これらの情報の集合を，**ファイル制御ブロック**(file control block)という．

上記のマクロに基づいて，ファイル制御ブロックで必要な情報の概略を示せ．

8. 上の問題の LIOCS を実現せよ．

第 3 章

一括処理オペレーティング・システム

注　　意

典型的な一括処理(batch processing)オペレーティング・システムを特徴づけることは，難しい．一括処理オペレーティング・システムが無いから難しいのではなく，典型的なものを見つけ出すことに問題がある．実際，多くの型の一括処理システムがある．ここでは，一括処理システムの2つの重要な型，すなわち，オフライン型とオンライン型に重点を置いて考えよう．

オンラインプログラミングとは，ユーザが，パンチカードやテープのような中間的記憶媒体を介さずに，自分のジョブ制御コマンド，プログラムの文，データなどを直接システムに入力し，また直接出力を受け取れることである．しかしながら，直接ジョブを入れ結果を受け取るためには，ユーザは，端末を使って通信回線を介してシステムにアクセスしなければならない．一方，**オフライン**の環境では，ジョブを，カード(あるいはテープやディスク上のカード像)のデック (deck) として提出し，それをオペレータが処理しカード読み取り機(またはテープやディスク装置) により読み取る．出力は，印刷機により紙へ(テープやディスク装置によりテープやディスクへ)印刷され，オペレータがユーザに分配する．ジョブの 入力/出力 のための 読み取り機/印刷機 は，通信回線を介してシステムから物理的に離れたところにあってもよいけれども，ユーザは，自分のジョブをデックとして，結果をリスティングとして扱う必要がある．この場合，遠隔ステーションをもつオフライン一括処理システムをもっているに過ぎない．

オフライン一括処理システムの中では，プログラミング言語サブシステムの体系が広く使われ，簡単に構成もできる．この型のシステムを，ある一般的な指針と詳細な概念とを使って，特徴づけしてみようと思う．

オンライン一括処理システムは非常に大規模で複雑さも大きいので，その構成や機能の良い抽象化ができないが，それがないと，良く理解することができなくなる．しかしながら，前と同じように，抽象化を通して，その機能や構成に重点を置くように努めよう．

この章で使う術語は，決して標準的なものではない．実際，特に一括処理システムには，標準的な語彙術語はない．一方，ここで述べる概念や構造は，一括処理システムで共通に見られるものである．

3.1 は じ め に

一括処理システムは，**ジョブの自動化** (job automation) を目的として開発された．ジョブの自動化とは，システムの処理量を上げるために，プログラムやデータから構成されるジョブを準備し，投入し，実行し，終了するための自

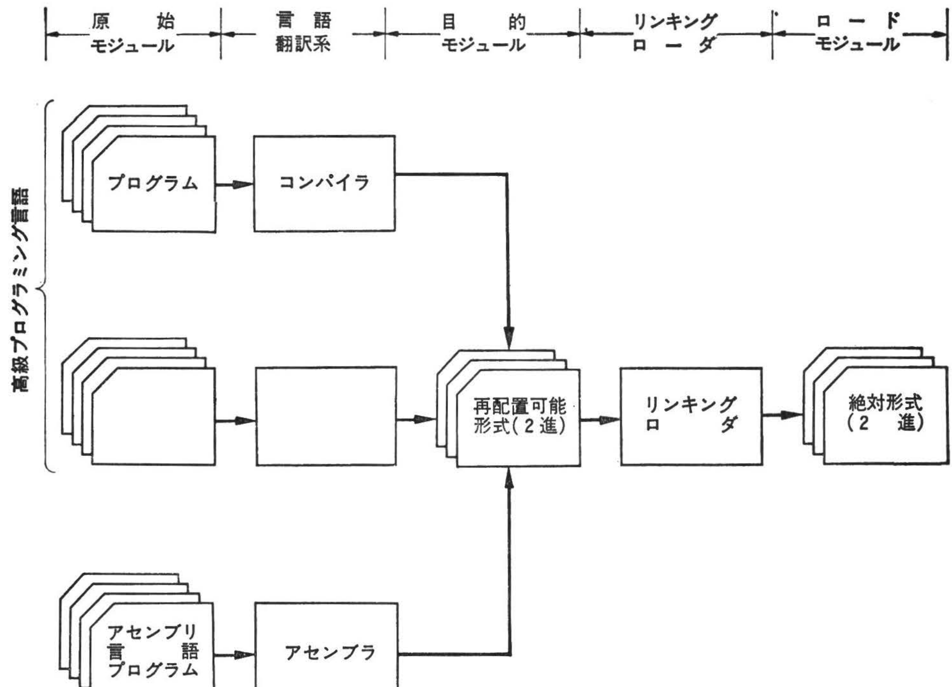


図 3.1　プログラミング言語システム

動的手続きを意味する.

たとえば，図1.1(a)に示したプログラミング言語システムをもう一度上に示すと，ユーザジョブを処理するために，次に示すステップを手続きとして使用する.

1. ジョブの原始モジュールで使うプログラミング言語を決定する.
2. そのモジュールのコンパイルあるいはアセンブルのために，適当な言語翻訳系を選択する.
3. コンパイルあるいはアセンブルの誤りなどの条件によるすべての異常終了の処理をする.
4. 目的モジュールを保存することにより，コンパイルやアセンブルの正常完了を管理する.
5. 言語翻訳系からリンキングローダへの遷移を監視する.
6. リンキングローダへの入力として，目的モジュールを準備する.
7. 外部記号やモジュールの欠落などの条件によるすべての異常終了を処理する.
8. ロードモジュールを保存することにより，結合の正常完了を管理する.
9. 実行のために，ロードモジュールおよび付随したデータをロードする.
10. 実行モジュールが生成する出力を分配する.
11. プログラムチェックや実行時間超過などの条件による異常終了を処理する.
12. たとえば，ジョブ料金計算の情報を更新したりして，正常完了を管理する.
13. 次のユーザジョブのために，ステップ1. へ戻る.

一括処理システムの生れる前には，オペレータ(又はユーザ)がこの手続きのステップを実行していた．ステップ1. に示してある原始モジュールのデックで使っている言語を決定するためには，人間がそのジョブに含まれるモジュールを調べた．オペレータは，翻訳系のロードモジュールを構成するカードのデックを持ってきて，適切な翻訳系を選んだ．そして，両方のデックを主記憶にロードした．そこで，ステップ2. に示したように，翻訳系の実行が行われる．翻訳系が生成する目的モジュールをカードにパンチしたときには，オペレータは，(カードパンチ機の)カード置き台からそれを取り出さなければならない.

新しくパンチされた目的モジュールを，入力のカード置き台に移し，リンキングローダのロードモジュールのデックと一緒に，主記憶に読み込む．そこで，リンキングローダの実行が始まる．これらは，ステップ3.，4.，5.，6.で要請される．再び，ユーザプログラムのロードモジュールを構成するパンチしたデックを，カード置き台から取り出し，入力カード置き台に移し，ユーザのデータカードと一緒に，実行のために主記憶に読み込む．これらは，ステップ8.と9.である．最後に，ユーザプログラムの実行が完了すると，オペレータは，次のプログラムとデータの実行の準備をしなければならない．

オペレータによるこのような手続きでは，システムの処理量を改善することは殆んどできない．デックを調べ，取り出し，移し，処理している間は，計算機システムは空いている．目的およびロードモジュールを格納する中間媒体としてパンチカードを使用するのは，計算機システムが磁気テープやディスク記憶を備えているときには不必要であるばかりでなく，わずらわしい．カードリーダ/パンチのような単位レコード機器は遅く，パンチカードは，人手で扱うのには繊細すぎる．したがって，一括処理システムの目的は，前述した手続きをオペレータが実行することを肩代りし，システムのよりよい処理量を確保することにある．

前章では，システムの処理量をよくするのは，CPU，チャネルおよび入出力機器をもっと有効に利用することによって達成できることを学んだ．有効な利用をするためには，CPU と入出力の並行的利用が不可能な場合，ある程度の入出力の空きを犠牲にしても，常に CPU を使用中にしておくことがよい．高速の CPU と大きな記憶があるときは，オペレータがジョブの処理を，人手で準備し，ロードし，監視しても，ハードウェアの能力に常に応じることはできない．さらに，ジョブの反復的仕事を人手で操作することは，事故を起こし易い．また，システムの処理量を改善する並行処理や割り込み処理は，IOCS や割り込み処理系など注意深く開発したシステムプログラムが一番上手に処理できる．問題が複雑なので，オペレータが一時的な意志決定をすることを許さないからである．したがって，ジョブの準備，投入，実行，終了に対する自動的手続きが計算機システムの“組み込み”プログラムによって遂行されることは，驚くことではない．一括処理システムは，システムの処理量を改善するジョブの自動化を提供する“組み込み”プログラムなのである．しかし，処理システムのジョブの自動化の程度は，システムで使える主記憶および二次記憶の大きさと，CPU チャネル，機器などのハードウェア要素の速度にほとんど制限されてしまう．

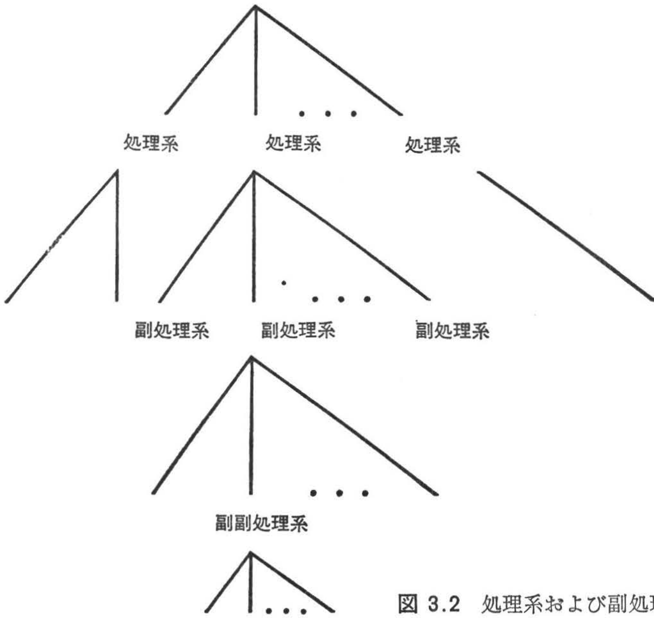


図 3.2　処理系および副処理系の階層

この章では，制限された記憶(8 Kバイトの主記憶と一つの二次記憶のディスク)をもつ計算機で走る単純なシステム，および，大きな主記憶(32 K 語の主記憶と，ディスクおよびテープの集団）を必要とするもっと複雑なシステムとを紹介しよう．また，結合によって，ジョブの自動化およびシステムの処理量を改善する方法を述べよう．

3.2　処理系および副処理系の階層：オフライン一括処理オペレーティング・システムの見方

オフライン一括処理オペレーティング・システムの一つの単純な見方は，それを処理系と副処理系の階層と見倣すことである(図 3.2 参照)．階層のどこのレベルにおいても，そのレベルの処理系は，あるタスクの制御と実行をする．一括処理オペレーティング・システムの階層に特有の興味ある現象は，同一レベルの処理系は，同時にお互の仕事を実行できないことである．さらに，同一レベルの副処理系と通信をする副処理系は，階層内におけるその副処理系の直前の処理系を介して通信をしなければならない．以上の議論より，階層内の任意の処理系は，次の事ができると考えられる．

1. 与えられたタスクに対し，どれかの副処理系を選択すること．
2. その副処理系から，別の副処理系への遷移を監視すること．
3. アクティブな副処理系に対して通信機能を提供すること．
4. アクティブな副処理系の異常終了による例外条件の処理をすること．
5. 自分が副処理系になっている処理系へ戻ること．

勿論，上述した処理系の実行は，その副処理系，そのまた副副処理系へと適用される．処理系に対して，次のことが重要である．

1. 適切な副処理系を選ぶためには，階層の同一レベルの他の処理系から伝えられる情報を使う．
2. 一つの副処理系から他への遷移の監視は，選択した副処理系のロード，初期設定，制御，終了を行なうことも含む．
3. 副処理系間の通信は，それらが異なった時間に実行されるので，すべての副処理系が常に知っている専用の領域を介して行なってもよい．
4. 例外条件により，処理系は，副処理系の詳細な実行系列の変更をする必要に迫られる．
5. 上位の処理系へ正常または異常状態で戻るときには，処理系は，戻りに関する何らかの情報をそのレベルのすべての処理系が知っている領域に格納する必要がある．

システムの階層における選択，監視，通信，変更などの役割をする処理系とは，一体何であろうか．いろいろのソフトウェアおよびハードウェア要素の明確な定義と記述とを，階層的な処理系という観点から与えれば，全体のシステムの構成は，すぐに求まるであろう．しかし，すでに存在するシステムの要素の定義や記述を処理系の階層として与えるのは，面倒なことである．実在するシステムの定義や記述は，膨大なものであり，ここでは不適当であるだけでなく，実在するシステムで階層の概念に基づいて設計したものは少ない．しかし，オフライン一括処理システムに対する階層的方法論は，わかり易い．階層の最下層には，算術演算器，インターバルタイマ，チャネルやCFUなどのハードウェア要素がある．それらのハードウェア(副)処理系が共通にもつ処理系は，割り込み処理系である．割り込み処理系は，算術演算器から桁あふれおよび下位桁あふれの信号を，インターバルタイマからタイマあふれの信号を，入出力操作実行系から入出力に関するチャネルの信号を，CPUからプログラムチェック(たとえば不当番地付け)やスーパーバイザコール(たとえば入出力操作の停止)の形を介して，情報を受け取る．選択者あるいは監視者として，割り

込み処理系は，割り込みを受け取る（すなわち，信号のチェックや呼び出しをする）と，次に動かすべき副処理系を決める．たとえば，割り込みがタイマあふれ信号のとき，タイマあふれ処理のルーチン(副処理系)が用意されていなければ，割り込み処理系はその信号を無視するし，もしルーチンが用意されていれば，(まだ主記憶に入っていないとき)それをロードし，初期設定をし，実行をする．タイマあふれの処理が完了するか異常終了したときは，そのルーチン(副処理系)から，割り込み処理系(処理系)に戻って終了する．この場合，割り込み処理系は，タイマのようなハードウェア要素，および，タイマあふれ処理ルーチンのようなソフトウェアモジュールの副処理系から構成されるソフトウェア処理系である．ここでは，オフライン一括処理システムを，処理系の階層として概略を述べよう．下層に関する膨大で詳細な記述を避けるために，階層の根から始めてシステムの記述を進めてゆこう．

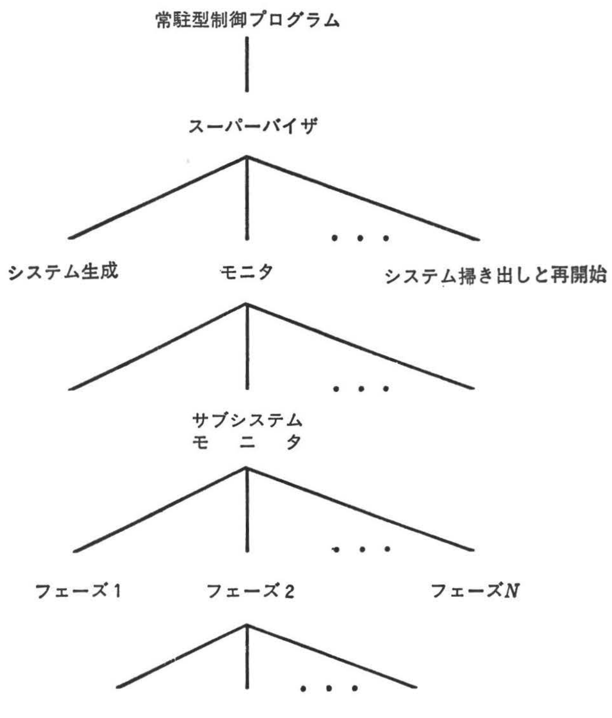


図 3.3 オフライン・オペレーティング・システムの階層的構成

3.2.1 プログラミング言語処理系のオフライン一括処理オペレーティング・システム

いろいろのプログラミング言語で書かれたユーザプログラムをオフラインで一括処理することは，伝統的に，典型的な一括処理オペレーティング・システムの主要な機能と考えられてきた．ここでは，図3.3に示すように，システムを多くのプログラミング言語副処理系（たとえば，その一つの副処理系はアセンブリ・システム）の根があるラベル付き階層と見做してみよう．

階層の根から，システムの処理が始まる．図3.3を見ると，階層の根は，常駐型の制御プログラムであることがわかる．プログラムの集団を一括処理するためには，下層の処理系を呼び，さらに今度はそれがどの副処理系が必要かを判断する．節点のラベルは，処理系の機能をあらわしている．これらのラベル付き処理系を，その機能の点から詳しく調べてみよう．

3.2.1.1 階　層　の　根

多くのサブシステムやプログラムに標準的基本入出力機能の集合を使えるようにする入出力制御システム（IOCS）の開発は，常駐型制御プログラムの練習問題になる．たとえば，IOCSの入出力操作実行系は，データチャネルを使用中の状態に保っておくためにそれを常に監視していなければならないので，常時主記憶の中にある．データチャネルと入出力動作を監視していれば，入出力操作実行系は，データチャネルのスケジュールをし，誤りの回復を実行し，コード変換をし，正常または例外的な戻りを実行することがいつでも可能となることを思い出そう．さらに，入出力操作実行系で必要とする装置制御ブロック，機器制御ブロック，記号装置名表などの制御ブロックも，主記憶の中になければならない．そこには，入出力操作の進行およびチャネルと機器の使用に関する状態の情報が，他のものと一緒に格納してある．この情報は，常に保守され，かつ，使用可能でなければならない．これらの理由のために，通常，低番地部の端の記憶の部分を，入出力操作実行系とそれに伴なう制御ブロックに割り付ける．

3.2.1.1.1 常駐型制御プログラム

主記憶に常に存在するスーパーバイザプログラムを，**常駐型制御プログラム**(resident control program)と呼び，それに伴なうデータ，作業領域および通信用領域を**核**(nucleus)と呼ぶ(**図3.4**の記憶配置参照)．常駐型制御プログラムと核とは，すべての副処理系およびユーザプログラムにとって重要なもので

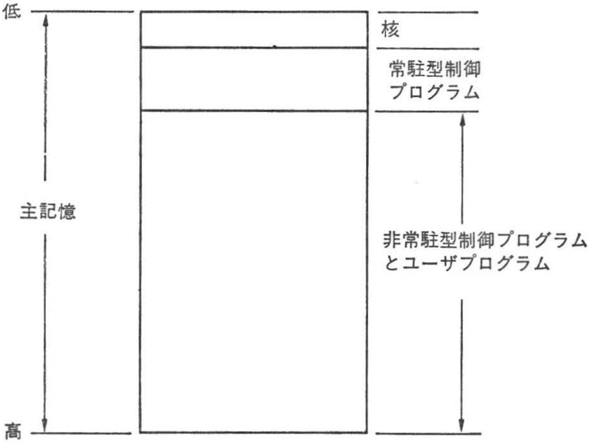


図 3.4　記憶の配置

あるから，その記憶領域は，ユーザプログラムのアクセスから常に保護してある．（初期の計算機および主記憶の制限された現在のミニコンピュータは，常駐型制御プログラムも記憶保護機構ももっていない）．

常駐型制御プログラムは，常に主記憶の中にいるので，規模が小さく，数も少ないものでなければならない．そうでないと，副処理系およびユーザプログラムのために使用できる残りの記憶が，非常に限られてしまうだろう．したがって，常駐とするシステムの機能は，次のことを行なうために必要なものだけとする．

1. スケジュールしていない要求，割り込み，割り出し(trap)† に応じる．
2. 副処理系（たとえば，プログラミング言語監視システム）を初期設定し，維持し，終了させる．

スケジュールしていない要求，割り込み，割り出しに応じるために，常駐型ルーチンは，

a) オペレータの要求，
b) 実行しているプログラムからのサービスたとえば入出力操作の要求，
c) 算術演算の桁あふれ，下位桁あふれ，チャネルの信号，不当命令，番地付け誤り，インターバルタイマあふれ，機械チェックなどの機械の割

† **訳注**　割り込みは実行しているプログラム以外の要素が直接の原因となって生ずる割り込み，割り出しは CPU で実行しているプログラム自身が直接の原因となって生ずる割り込みをさす．前者を外部割り込み，後者を内部割り込みともいう．

り込みあるいは割り出し，

を処理できなければならない．

いくつかの場合には，その処理ルーチンが，要求，割り込み，割り出しを直接サービスしないかもしれない．それが生じた条件を単に解析し，そのような条件をさらに処理するのが正しいかどうか決定する．要求をさらに処理する必要があるならば，処理ルーチンは，その要求のサービスをするために，適切なルーチンを二次記憶から核またはそれ以外の作業領域にロードしてくる．

入出力操作の場合，処理ルーチン(たとえば，入出力実行系)は，すでに主記憶の中に存在する．したがって，入出力要求とチャネル割り込みは，すぐに処理できる．

システムを初期設定し，維持し，終了させるために，常駐型制御プログラムは，システム機能のロードをひき起せるようなルーチンを提供しなければならない．オペレーティング・システムの最も重要な機能は，ユーザによって提示された処理の要求の型を決めることである．これらの要求に基づいて，望んでいる処理に対する以後の副機能を初期設定することができる．**スーパーバイザ**(supervisor)は，そのような機能の一つとして，一つのユーザのジョブ(たとえば，ユーザの連続したプログラムとデータの集団に対するすべての仕事)から別ののジョブへ遷移することを監督する．スーパーバイザの制御の下にある副機能は，次のようなものである．

a)　**モニタ**(monitor)——一つのサブシステム（たとえば，FORTRAN プログラミングシステム)から他のサブシステム(たとえば，アセンブリシステム)へ遷移することを監視する．

b)　**システム掃き出しと再開始** (system dump and restart)——後ほどリストをとったり実行したりするために，主記憶の情報をシステムのユーティリティ装置(たとえばディスクやテープ)に退避する．

c)　**システム生成**——新しいシステムを生成したり，現在のシステムを編集したり，写しを作ったりする．

a）について考えると，各サブシステムは，**サブシステムモニタ**(subsystem monitor)と呼ぶ自分自身のモニタをもっており，それは，そのサブシステムの一つのフェーズ（たとえば，FORTRAN システムの入力プログラム走査フェーズ)から，同一サブシステムの別のフェーズ(たとえば，FORTRAN システムの記憶割り当てフェーズ)へ遷移することを監視する．

上述したシステム機能の実際のロードは，**システムローダ**(system loader)

が行なう．このため，システムローダも常駐型とする．すなわち，システムローダは，この節の前半で述べた常駐型制御プログラムの第2の機能を実行する部分になっている．

システムローダの機能は，次の通りである．

1. 該当するシステム機能を含んでいる物理レコードの先頭に，入出力機器を位置づけする．
2. レコードを読み込み，検証する．
3. システム機能の次の部分に含まれるレコードの先頭に，機器の位置づけを開始する．
4. いまロードしてきたレコードに，制御を渡す．

システムローダは，アセンブリシステムの章で述べたリンキングローダが持つべき6個の機能はどれも持っておらず，ロードモジュールとなっているプログラムをロードするだけである．その機能は単純であるから，システム機能のロードは高速でかつ最小のオーバーヘッドがかかるだけである．

3.2.1.1.2　核

制御ブロックと作業領域に加えて，核は次のものを持っていてよい．

1. サブシステム間の通信用の領域，すなわち，制御しているサブシステムの名前，制御しているサブシステムのフェーズの名前，実行のために割り当てた時間，ロードモジュールが使える主記憶の制限量などを含む．
2. 現在の日時などのシステムのデータ．
3. 機械の機能に対して割り付けた固定の番地，すなわち，割り込みや割り出しの入口（たとえば，チャネル割り込み）や飛び込み先（たとえば，入出力操作実行系）．
4. いろいろの表や制御ブロックへのポインタ．
   a) チャネルによる装置制御ブロックへのポインタ，
   b) チャネルによる機器制御ブロックへのポインタ，
   c) システムユーティリティの装置へのポインタ，
   d) システムライブラリの内容表へのポインタ，

a) および b) は，入出力操作実行系がチャネル動作のスケジューリングの際に使う．c)は，アセンブラやコンパイラのサブシステムと，記憶掃き出しおよび再開始のシステム機能が使う．d)は，ローダがシステム機能やサブシステムのフェーズをロードするときに使う．

3.2.1.2　階層のほかの部分：非常駐型制御プログラム

スーパーバイザ，モニタ，サブシステムモニタ，掃き出しおよび再開始の大きさをすべてあわせると，これらをすべて主記憶にずっと置いておくのは実際的でないことがわかる．そのかわり，これをディスクやテープのような二次記憶機器に格納しておく．これらのシステム機能をロードすることができるように，システム機能をそれぞれ全部アセンブルし，絶対形式になるように再配置しておく．さらに，いくつかのフェーズから構成されているサブシステムの場合は，各フェーズは，別々にアセンブルし再配置する．これらを集めて，**システムライブラリ装置** (system library unit) とよぶシステムユーティリティに標準的な形式で格納する．

システムライブラリ内容表 (system library's table of contents) とよぶ次に述べる形式をした2層の表を作り，システム機能やサブシステムのアセンブルや再配置をするとき**システムライブラリ更新サブシステム** (system library update subsystem) がこれを保守する．ここで，

A_i は，サブシステム (たとえば FORTRAN) あるいはシステム機能 (たとえば DUMP) の名前，

p_i は，サブシステムあるいはシステム機能 A_i に関する情報のブロックへのポインタ，

$p_i{}^j$ は，A_i の j 番目のフェーズの名前，

$Q_i{}^j$ は，フェーズ $p_i{}^j$ をロードすべき主記憶の番地，および，フェーズ $p_i{}^j$ を含む物理レコードの先頭の二次記憶上の番地に関する情報，

である．

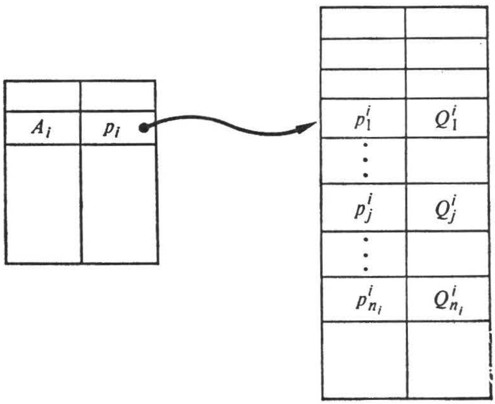


図 3.5　システムライブラリ内容表

さらに，表にあらわれる p_i^j の順番は，フェーズのロードの順序を決める．すなわち，サブシステム A_i（たとえば，FORTRAN）に対して，フェーズ p_1^i（FORTRAN の原始文を分類し，診断情報を作り，記号ディクショナリを作り，文を内部テキストに変換する入力プログラムの走査フェーズ）のロードが最初で，

次に，フェーズ p_2^i（定数を生成し，記憶を確保し，同値のものを生成し，FORMAT 文を翻訳する記憶割り当てフェーズ），

次に，フェーズ p_3^i（算術および論理式を計算する順序に並べ換える算術および論理翻訳フェーズ），

次に，フェーズ p_4^i（添字の使用の最適化をし，インデックスレジスタを割り当てる添字解析フェーズ），

次に，フェーズ p_5^i（実行可能な機械命令を生成する命令生成フェーズ），がくる．

3.2.1.3 コマンド言語

コマンド言語 (command language) あるいは**ジョブ制御言語** (job control language) と呼ぶ言語を，オペレーティング・システムとの通信のために使う．FORTRAN，COBOL やアセンブリ言語のようにコンパイル，アセンブルや再配置が必要となるプログラミング言語と違って，コマンド言語は，本質的に解釈的(interpretive)言語である．基本的には，コマンド言語の文は，コマンドの名前とそのコマンドに対する引数の集合とから成り立っている．文の解釈をする過程は，次のステップに従う．

1. 文を読み，退避する．
2. 正当なコマンド名からできている表のエントリと比較して，コマンド名の検証をする．
3. コマンド名が不当である場合，すなわち，表内に見つからない場合，誤り処理ルーチンに飛ぶ．
4. そのように名づけたコマンドを実行するシステムのルーチンへ飛ぶ．必要としているシステムルーチンは常駐していないかもしれないので，解釈系は，そのルーチンをロードするためにシステムローダを呼び出す．システムローダは，システムライブラリ内容表中でそのルーチン名を探して，実行のためにシステムライブラリからルーチンをロードする．
5. そのルーチンに引数を渡す．

6. ルーチンがその処理を終了したら，1. へ戻る.

オペレーティング・システムは階層構造をしているので，ジョブの最初のコマンドは，スーパーバイザのロードを引き起せるコマンドでなければならない. スーパーバイザが得られれば，次からのコマンドは，スーパーバイザのもつ解釈系で処理しなければならない. 不当なコマンドがあったときには，スーパーバイザは，ジョブの性格に応じてそのコマンドを飛ばすかジョブを終了させる. モニタを持ってくるためには，スーパーバイザが理解できる適当なコマンドを出さなければならない. モニタが得られれば，適当なコマンドを使って，サブシステムを持ってくることができる. 通常，コマンドの名前は，サブシステムの名前を含んでいる. モニタの理解できないコマンドがあらわれたときには，そのコマンドを飛ばすか，スーパーバイザへ制御を戻す. サブシステムが得られたならば，サブシステムモニタが，それ自身のコマンドを解釈する. 不当なコマンドがあれば，モニタに制御を戻す.

要約すると，オペレーティング・システムが階層構造になっていると，階層の各レベルにおける制御システムは，自分自身の解釈系と正当なコマンド名の表とを持たなければならない. 制御は，階層の直ぐ上のシステムへ戻る. 物理的には，一つの解釈系を制御プログラムが共有し，正当なコマンド名の合同表は，簡単にアクセスするためにシステムライブラリの中に格納されているかもしれないけれども，スーパーバイザへの正当なコマンドがサブシステムモニタにより解釈されれば，スーパーバイザはサブシステムを終了させる. すなわち，正当なコマンドの表にあるコマンドは，適切な解釈をするために階層の正しいレベルで使わなければならない.

コマンドは，いろいろのシステムの要素やサブシステムが提供できる機能に関連があるので，それらの機能を知ればコマンドの守備範囲が予想できる. したがって，可能なコマンドの一覧表を編集するかわりに，スーパーバイザ，モニタやサブシステムモニタの役割を論じよう. また，3.2.3.2 節では，スーパーバイザ，モニタ，サブモニタを働かせるための例をいくつか示そう.

3.2.2 オフライン・オペレーティング・システムの構成要素

3.2.2.1 スーパーバイザ

通常，スーパーバイザが主記憶中にないときには，そのロードをひき起こせるコマンド(たとえば，JOB コマンド)から，一つのジョブが始まる. したがって，スーパーバイザは，ユーザのジョブを初期設定するためにシステムライブラリから主記憶内に持って来られるオペレーティング・システムの第一の“構

成要素”である．スーパーバイザが一度主記憶に入れば，ジョブの初期設定が始まり，次のことを行なう．

1. ユーザのジョブ料金計算の検証と更新．JOB コマンドが見つかると，スーパーバイザは，直ちにシステムの料金計算ルーチンに制御を渡す．料金計算ルーチンは，ユーザの識別名，使用可能な金額，ジョブの型とクラスをチェックし，ログイン(log-in)手続きを行なう．
2. 時間の割り当て．ジョブに割り当てられた時間を，核内の制御情報の一部としてインターバルタイマに記録する．
3. 入出力構成の確立．前に述べたように，入出力構成は，装置および機器制御ブロックの中，および，記号装置名表の中で定義されていた．スーパーバイザは，この制御ブロックと表とを保守しなければならない．一つのジョブに対して，スーパーバイザは，主記憶の中に持って来る最初で最後の要素だからである．一方で，スーパーバイザは，ユーザが入出力装置を柔軟に使いこなすために，コマンドを介して構成を変更することを許す．また一方で，スーパーバイザは，ユーザが入出力構成について配慮しなくても済むように，標準的構成を維持している．

   スーパーバイザの提供するサブルーチンは，次のようなコマンドの処理ができる．

   a) ユーザに対し，ある装置を確保すること．
   b) ユーザに対し，ある装置を使えなくすること．
   c) 2つの装置を交換すること．
   d) テープの巻き戻しやアンロード(unload)のようなある種のファイル終了の機能を実行すること．
   e) 標準的入出力構成を再確立したり，オペレータに対し現在の入出力構成を出力すること．
4. システム機能の開始と終了．これらは，掃き出しと再開始，および，特定のジョブ処理の要求を監視することを含む．
5. 常駐型制御プログラムに戻ること．ジョブの終了と共に，スーパーバイザは，ジョブ終了のコマンドかまたはオペレーティング・システム自身によって呼び戻される．どちらの場合にも，スーパーバイザは，ジョブに対して確保した装置を解放し，ジョブをログアウト(log-out)し，次のジョブを開始する．ある場合には，スーパーバイザは，次のジョブのために標準的入出力構成を再確立する．

3.2.2.2 モ　ニ　タ

スーパーバイザは，ジョブの開始と終了とを扱うのに対して，モニタは，主としてサブシステムあるいはオペレーティング・システム機能の開始と終了とを扱っている．与えられたオペレーティング・システムのもつサブシステムやシステム機能の数は，構成に応じて異なっているけれども，基本的な型は，次のものと考えることができる．

1. 言語処理系――FORTRAN, COBOL，アセンブラ，リンキングローダなど
2. システムライブラリ更新
3. 言語ライブラリ更新
4. 整列/併合
5. ユーティリティ
6. ユーザプログラム実行フェーズ

サブシステムを開始する前に，モニタのすべき主な機能は，それを実行するために何か基本的なシステム機能を使わなければならないかどうかを決めることである．たとえば，システムライブラリ更新サブシステムは，システムライブラリのプログラムを，物理レコードの形式で処理する．これは，レコードのブロッキングおよびデブロッキングのために，入出力バッファリング・サブシステム(IOBS)の助けを必要とはしていない．一方，コンパイラやアセンブラの言語処理系は，IOBSの存在を必要とするような入出力操作を使ってもよい．したがって，モニタは，最初の場合，IOBSのロードを禁止し，後の場合，IOBSの持ち込みを開始する．

サブシステムが終了したときには，モニタは主記憶領域を空にし，次のサブシステムを開始する．

3.2.2.3 サブシステム

3.2.2.3.1 システムライブラリ更新サブシステム

前にも述べたように，システムライブラリは，絶対形式のシステムプログラムとサブシステムとから構成されている．システムライブラリ更新サブシステムは，システムライブラリ中のシステムプログラムやサブシステムのロードモジュールを加えたり，修正したり，消去したりするために使う．

サブシステムへの入力は，いろいろの言語や形式(たとえば，FORTRAN原始言語形式，再配置型2進形式，絶対2進形式，システムライブラリの物理

レコード形式など)であり，また，コンパイル，アセンブル，あるいは入力やライブラリの編集をするために多数の装置を使用するので，動作を見張るために，次のようなサブシステムモニタが必要になる．

システムライブラリ更新サブシステムのモニタは，次の事を行なう：

1. システムライブラリの追加，削除，修正を制御するコマンドの処理をする．
2. どの入出力装置を使うのかを決め，それらを初期設定する．
3. システムライブラリを含んでいる旧システムライブラリ装置および新システムライブラリを作り出す新システムライブラリ装置のラベルの検査をする．
4. 言語ライブラリ内に再配置形サブルーチンとしてプログラムが格納してあれば，それらを見つけ出す．
5. 入力プログラムが未だ絶対形式になっていなければ，言語処理系サブシステムとリンキングローダを呼び出し，絶対形式のプログラムを作らせる．
6. 更新の実行に必要なフェーズの実行を指示する．

システムライブラリ更新プログラムは，次のフェーズをもっている：

1. 入力レコードのブロッキングと整列とをして，入力レコードがシステムライブラリのフェーズとなるような順序になることを確認する．
2. 上のフェーズに対し，システムライブラリ内容表に新しいエントリを作ったり古いエントリを修正したりする．
3. 新しい内容表をユーティリティ装置上に書き出す．
4. 古いシステムライブラリ内のフェーズと，新しく作成したフェーズとを併合する．
5. 新しいシステムライブラリを，ユーティリティ装置上に書き出す．

3.2.2.3.2　言語ライブラリ更新サブシステム

このサブシステムの機能は，前節で述べたものと似ている．大きな違いは，言語ライブラリにあるプログラムは，絶対形式でなく再配置型の形式でサブルーチン(もっと詳しく言えば，コントロールセクション)として格納してあることである．このため，言語ライブラリは，しばしば**サブルーチンライブラリ**と呼ばれる．一般に，言語処理系サブシステムは，それぞれ自分自身のサブルーチンライブラリをもっている．したがって，たとえば，FORTRAN コンパイラサブシステムは，FORTRAN サブルーチンライブラリをもち，アセンブリ

サブシステムは，マクロ定義ライブラリをもっている．しかし，実際には，これらのライブラリは物理的に結合され，まとめて，言語ライブラリまたは再配置型ライブラリと呼ぶ．概念的には，2層のシステムライブラリ内容表を構成するために使ったものと同じ技法を拡張して，**言語ライブラリ内容表** (language library's table of contents)を構成する．

サブルーチンは，いくつかのコントロールセクションから成り，異なったサブルーチンが同一のコントロールセクションを共有してもよいから，言語ライブラリ内容表は，かなりの工夫を必要とする(**図 3.6** 参照)．

サブルーチンに必要なコントロールセクションをすべて見つけ出すために，最初にサブルーチン名から探索する．勿論，サブルーチンは，内容表 T_1 にあるコントロールセクション名 A_i である．A_i に付随して，2つの番地 p_i と q_i とがある．p_i が空であれば，このサブルーチンは，只一つのコントロールセクションから成り立ち，その居所は q_i である．p_i が空でなければ，サブルーチンを構成する m 個のコントロールセクション $A_1{}^i$, $A_2{}^i$, $\cdots$, $A_m{}^i$ のリスト L_i がある．この場合，内容表 T_2† の方を探す．p_i を利用すれば，リスト L_i の中の一つのエントリが見つかる．このエントリは，コントロールセクション名 $A_j{}^i$ と，T_1 内の $A_j{}^i$ の居所を示す番地 $r_j{}^i$ とからできている．T_1 内の $A_j{}^i$ は，やはり，前述したような2つの番地 $p_j{}^i$ と $q_j{}^i$ とをもっている．

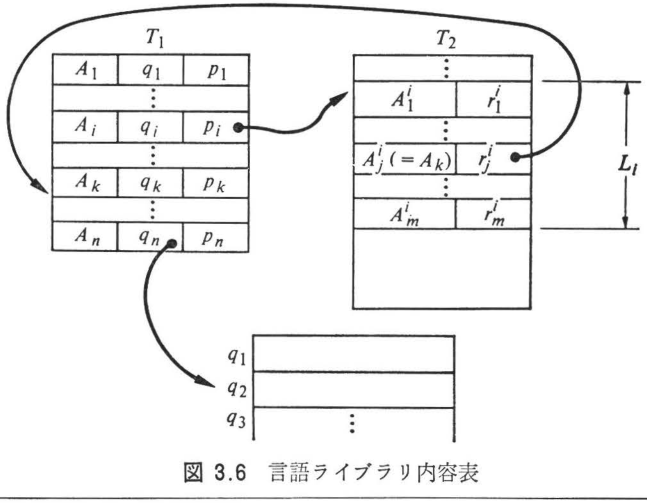


図 3.6 言語ライブラリ内容表

† p_i を先頭にしてリストを作るように，コントロールセクションをポインタ $r_j{}^i$ で結合しているので，(コントロールセクションの)数 m は，探索のために必要ではない．ここでは，議論を完結させるために，m を導入しただけである．

すべての $p_j{}^i$ が空であると仮定すると，サブルーチンに必要なコントロールセクションは，次の通りである．

q_i で A_i,
$q_1{}^i$ で $A_1{}^i$,
…
$q_m{}^i$ で $A_m{}^i$.

しかし，一般に，$p_j{}^i$ のいくつかは，リスト $L_j{}^i$ へのポインタである．この場合，上に述べた探索方式を繰り返し使って，空のポインタで終了するまで続ける．繰り返し型の探索方式は，循環的な探索を避け，多数のエントリをたどるという努力が必要になるけれども，従来からあるサブルーチンや内容表を再処理しなくても，プログラマは，追加のサブルーチン(すなわち，コントロールセクション)を加えて，従来からあるサブルーチン(またはコントロールセクション)に対する外部参照を行なうこともできる．

言語およびシステムライブラリ更新サブシステムは，そのサブシステムモニタの機能やプログラムが似ているので，通常一緒にして実現する．

3.2.2.3.3　言語処理系サブシステム

A.　アセンブラ

アセンブラについては，アセンブリ・システムに関する章で論じている．

B.　リンキングローダ

リンキングローダは，入力として一つの形式——再配置可能な形式——のプログラムを受け取り，出力として別の形式——絶対形式——のプログラムを作り出す．このために，それを言語翻訳系と考えることができる．大部分の翻訳系(たとえば，ALGOL や FORTRAN のコンパイラ)は，バッファの割り当てや機器やチャネルの使用のような計算機資源の有効利用に関して無関心であるけれども，リンキングローダはこれと異なり，この問題に直接取り組むただ一つの翻訳系である．すべてのコンパイラやアセンブラは，リンキングローダによってロードされる再配置型プログラムを作成するので，主記憶，機器，チャネルを有効に使用できれば，すべてのコンパイラやアセンブラにとって有益である．

主記憶の規模が制限され，再配置型プログラムが大きい場合には，リンキングローダをいくつかの実行フェーズに分割する．各フェーズは，フェーズから

フェーズへの遷移を監視するリンキングローダのサブシステムモニタが持ってくる．

フェーズ 1　入力変形フェーズ

テープやディスクなどの媒体上にあるローダへの入力を，**ロードファイル** (load file)という．一般に，ロードファイルは，再配置型プログラムの集合とプログラムに施される修正に関するコマンドとから構成される．ローダのフェーズ1では，プログラム自身と，次に述べる再配置型プログラムの制御情報とを識別する．

1. 各コントロールセクションに対して，外部記号ディクショナリと内部再配置型記号ディクショナリを含む記号ディクショナリ．
2. ファイル制御ブロック．
3. ローダに対するコマンド．

上述の制御情報は，圧縮した形式で主記憶内に格納される．残りの記憶は，再配置型プログラムそのものを格納するために使う．プログラム自身は，使用

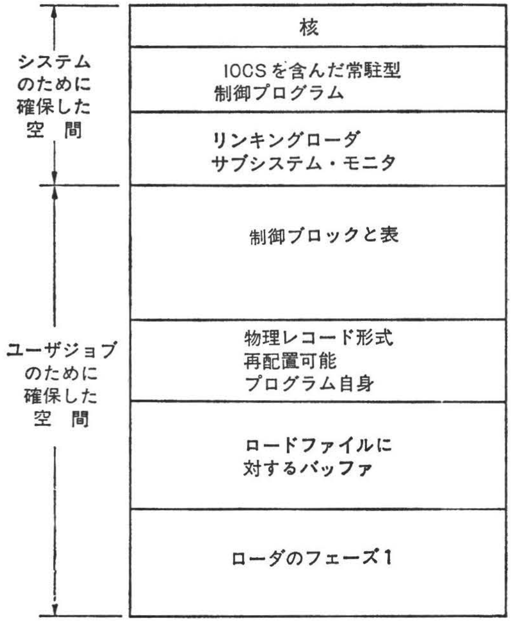


図 3.7　フェーズ1終了時の主記憶の論理的配置

可能な空間よりも大きいかもしれないので，フェーズ1は，プログラムを物理レコードに編成して，IOCSを通じてシステムユーティリティ装置へ書き出したりそこから読み込んだりする．こうすれば，プログラムの大きさは，ローダを使用中に制限を与えることはない．唯一の制約は，制御情報は充分小さくて主記憶の中に圧縮して格納できなければならないことである．フェーズ1の最後には，ファイル制御ブロックと記号ディクショナリの表とが作られる（図3.7参照）．制御ブロックや表が，割り当てられている空間以上のものを要求したときには，形式をととのえたプログラムそのものを，システムユーティリティのテープやディスクに掃き出して制御ブロックや表に対する空きを作る．

フェーズ 2　相互参照およびサブルーチンの解析フェーズ

ローダのフェーズ2の主な機能は，どのサブルーチンの集合をサブルーチンライブラリから要求するのかを決めることである．一つのサブルーチンは，複数個のコントロールセクションからできていてもよいし，異なったサブルーチンが同一のコントロールセクションを共有していてもよいので，フェーズ2は，サブルーチンライブラリの内容表を見て，必要なサブルーチンの集合を構

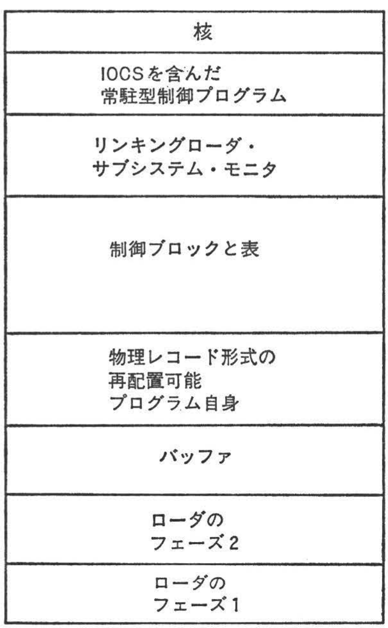


図 3.8　フェーズ2終了時の主記憶の論理的構造

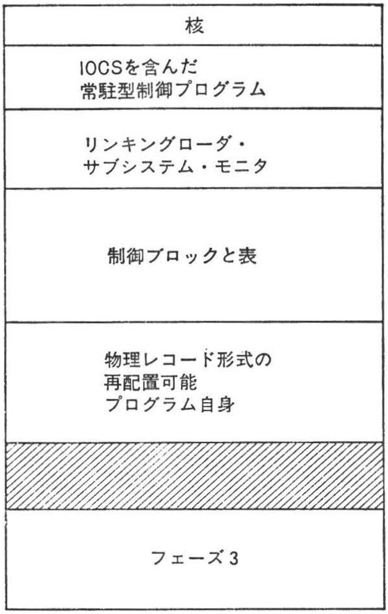


図 3.9　フェーズ3終了時の主記憶の論理的構造

成するコントロールセクションを決める．一度コントロールセクションが決まれば，フェーズ2は(まだ主記憶に残っている)フェーズ1を呼び出し，コントロールセクションを処理し，制御情報を圧縮し，それらを制御ブロックや表に挿入する．次に，ローダに対するコマンドを処理し，制御ブロックや表を更新する(図3.8参照)．

フェーズ 3　絶対番地の割り当て

ローダのフェーズ3(図3.9)の主な仕事は，次の通りである．

1. 制御ブロックと表に示されているように，コントロールセクション(プログラムそのものではない)の名前に絶対番地を割り当てる．
2. 再配置型プログラムによって要求されている記号装置に，物理機器を割り当てる．この割り当てにおいては，入出力操作とIOCSに関する章で述べたように，チャネルと機器の釣り合いのとれた使用を達成しようと試みる．
3. バッファの割り当て．前に述べたように，バッファ割り当てアルゴリズムは，入出力操作とIOCSに関する章で述べている．

フェーズ 4 プログラム自身の再配置フェーズ

制御ブロックと表における記号名と参照に対しては，すでに絶対番地が割り当てられているので，それをプログラム自身の再配置のために使う．すなわち，上述の情報に基づいて，絶対形式のプログラム自身を，その再配置型形式から生成することができる．しかし，プログラム全体が絶対形式に変換されるまでは，制御ブロックと表は主記憶の中になければならない．制御ブロックと表とは，主記憶の低番地の端に置いてある(**図 3.9** 参照)ので，フェーズ 4 を除いた上端の部分は，絶対形式のプログラム自身を格納するために使用できる．この領域が充分でないときは，絶対形式のプログラム自身をシステムユーティリティ装置へ掃き出す．再配置型プログラムのすべての物理レコードを処理すれば，フェーズ 4 は，プログラム自身をその絶対形式に再配置することができる．絶対形式のプログラム自身は，主として主記憶に格納してある．それが大きくなければ，ユーティリティ装置には掃き出さない．

今までの議論で明きらかであるが，リンケージロードを完了するために必要な時間は，使用するコントロールセクションの数，プログラムの大きさ，さらに他のユーザプログラムのために記憶の中に確保してある空間などに応じて，違ってくる．これは，システムの性能に対する単純な図式が何故存在しないかを説明するのに役立つであろう．

フェーズ 5 絶対形式のプログラム自身のロードと実行のフェーズ

フェーズ 5 は，絶対形式のプログラムを実行に移す最後の段階である．絶対形式のプログラム自身のセグメント(通常はコントロールセクション)は，連続したロードの順番に並んでいないかもしれないので，ある機械においては，適切な実行の順序に並べ直す必要がある．この場合，並べ換えは，セグメントを正しい順序になるように，分散して書き出すことにより行なう．プログラム自身を並べ換えたならば，プログラムの実行を始めてもよい(**図 3.10** 参照)．

C. コンパイラ

コンパイラを言語サブシステムの一つとしてオペレーティング・システムに組み込むために，コンパイラの製作者は，サブシステムモニタと実行フェーズの観点からコンパイラの構成を知る必要がある．多くの教科書がコンパイラの作製とシステムとの問題について扱っているので，ここでは，これ以上議論を進めることはしない．

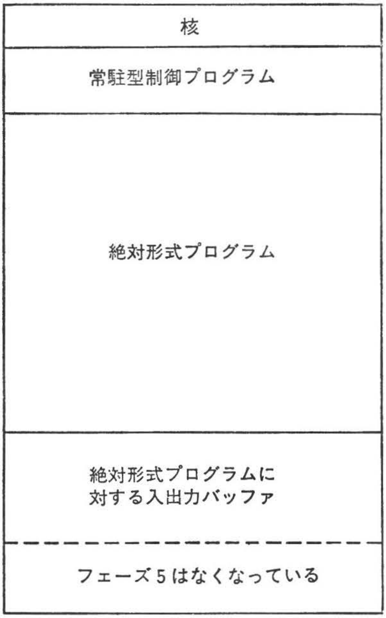


図 3.10　フェーズ5終了時の主記憶の論理的構造

3.2.2.3.4　プログラムフェーズ実行サブシステム

絶対形式のプログラムは，私用のファイルかまたはシステムの公共用ライブラリへ，永久的に退避しておくことができる．どちらの場合にも，ユーザやシステムプログラムをファイルやライブラリへ入れるための準備は，リンキングローダによって最初に行なわれなければならない．前節にも示したように，絶対形式のプログラムすなわちロードモジュールを生成するのは，リンキングローダである．しかし，一つのジョブが絶対形式のプログラムの実行をしなければならないときには，プログラムフェーズ実行サブシステムを持ってこなければならない．このシステムは，絶対形式のプログラムが常駐的なものであればライブラリやファイルから検索してこれるし，また，入力機器から絶対形式のプログラムを読んでくることもできる．コンパイラと同様に，大きなユーザやシステムプログラムに対しては，リンキングローダがそれを実行フェーズに編成して，記憶の圧迫されている環境で順序よく実行できるようにする必要がある．したがって，サブシステムモニタは，フェーズの名前を識別し，フェーズからフェーズへの遷移を監視し，フェーズの実行を終了させることができなければならない．

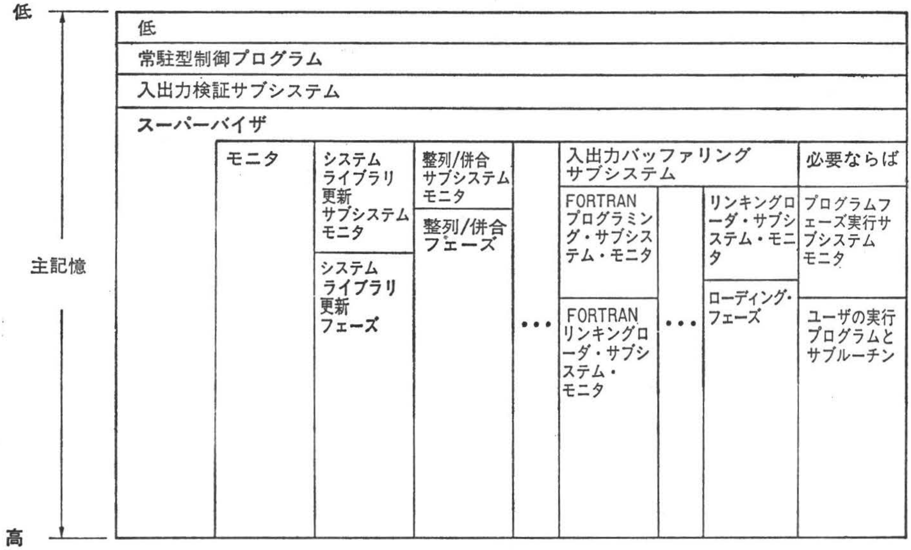


図 3.11 オフライン一括処理オペレーティング・システムに含まれるサブシステムの主記憶の論理的配置

3.2.2.3.5 ほかのサブシステムとシステムプログラム

ほかの重要なサブシステムやシステムプログラム，たとえば，整列/併合サブシステムや虫取り用システムプログラムなどについては，この教科書では扱わない．これらは，学生がもっと学習するために，課題として研究すべきである．

3.2.3 オフライン一括処理オペレーティング・システムの要約

前の節までに述べてきた一括処理システムは，一般的なものである（図 3.11 参照）．システムの特徴は，次の通りである．

1. すべてのサブシステムやプログラムに対して入出力操作を処理する集中型入出力制御の存在．
2. システムライブラリ，言語ライブラリ，私用ファイルに存在しているすべてのサブシステムやプログラムに対して，ラベルを作成したりそれを検証したりする能力．
3. 各言語処理系のすべてのサブシステムおよびサブルーチンを保守するために，システムおよび言語ライブラリ更新サブシステムを使用できること．

4. サブシステム間の通信とジョブの自動化のために，核，スーパーバイザおよびモニタの階層構造を利用すること．

このシステムは，基本的には，いろいろのプログラミング言語処理系やサブシステムの親システム(host system)として働く．

親システムとして，これは，いろいろのプログラミング・サブシステムに対して階層的構成をとるように強制する．ユーザがジョブ制御言語の使用を誤まると，オペレーティング・システムがプログラムやデータを時期尚早に終了させてしまうことが，しばしばあるであろう．オペレーティング・システムがもっと複雑な構成をしていれば，もっと多くの混乱が生ずるであろう．

ジョブ制御言語を正しく使うためには，プログラマは，オペレーティング・システムの構成を理解する必要がある．したがって，彼等には，問題向き言語だけでなく，システム構成とジョブ制御言語も教えなければならない．ジョブ制御言語の利用をするために，いくつかの改良が行なわれてきた．たとえば，(1) すべてのシステムコマンドは，サブシステムコマンドの先頭にくるどんな記号とも異なったもので始まる．(2) 頻繁に使用するコマンドの集合に置き換るマクロコマンドを使用する．しかしながら，大抵の場合，コマンド言語は，与えられたオペレーティング・システムの構成を見てから，後で考えて設計したものとなっている．将来のオペレーティング・システムの設計者は，コマンド言語の設計から始めて，それがオペレーティング・システムの設計に影響を与えたり指示をしたりするようにすれば，有益なものとなろう．

3.2.3.1 例

一般的なシステム階層に若干の簡略化を施して，伝統的な一括処理システム，IBM 360 Basic Operating System(BOS)を，処理系および副処理系の階層として記述することができる．

階層の根は，常駐型制御プログラムと核とから構成される．共通の入出力制御システム，すなわち，物理 IOCS(PIOCS)は，常に主記憶の中にある．もっと詳細に述べると，**スーパーバイザ**という常駐型プログラムは，次のルーチンから成り立っている．

1. PIOCS,
2. スーパーバイザコール割り込み処理ルーチン,
3. 外部割り込み処理ルーチン,
4. プログラムおよび機械チェック処理ルーチン,

5. オペレータ通信処理ルーチン,
6. フェッチルーチン.

2.から4.のルーチンは割り込みを処理するので, まとめて, **割り込み処理系**(Interrupt Handler)という. さらに, 専用のディスクから絶対形式のプログラムやデータをロードしてくるフェッチルーチンは, 絶対形式ローダである. これは, **システムローダ**(system loader)ともいう. そこで, スーパーバイザの4つの主な機能は, 入出力要求と実行の監視, 割り込みの処理, 非常駐型システム副機能のロード, および, オペレータとの通信である. 核は, 4つの基本領域から成り立っている.

1. 物理装置ブロックの表(PUB表). 使用している記号入出力装置ごとに, 表に一つのエントリがある. 通常は, 記号装置には, 物理装置が割り当てられる. しかし, ディスクのような直接アクセス機器の場合は, 2つ以上の記号装置が単一の機器に割り当てられる.
   表のエントリの位置は, そのエントリの意図した記号名を意味している. たとえば, 2番目のエントリは, 常に, ジョブ制御カードに対するカード読み取り機を表わす. 各エントリは, 割り当てた機器を特徴づける4つの情報から構成される. すなわち, 機器がつながっているチャネル, 一意的な機器識別番号, 機器の型, 記録密度と仕様の4つである.
2. 通信用領域. この領域内には, 日付や機器構成など, システムに関するデータを記録する. 制御しているプログラムの名前や, 実行しているプログラムのフェーズの名前など, システム間通信用の領域もある.
3. 機械の機能, たとえば, タイマやユーザのタイマ処理ルーチンへの戻り場所などに割り当てた固定の場所.
4. 論理入出力パッケージ (OPEN や CLOSE など), システム掃き出し, 再開始, チェックポイントルーチンなどをロードし実行するための一時的領域.

IBM 360/BOS の核は, 前節に述べた一般化したシステムの核が持っている情報と同じ型のものから構成されていることがわかる(図3.12参照). 唯一の違いは, 制御ブロックの扱いにある. IBM 360/BOS では, 物理機器と記号装置に関する制御情報は, 核の中の物理装置ブロックの表の中にある. 一方, 一般化したシステムでは, バッファリングやラベルの検証のようなもっと精巧な論理的入出力動作を実現できるので, もっと凝った制御ブロックが必要になる. これらの制御ブロックは, かなり大きいものである. OPEN, CLOSE, GET,

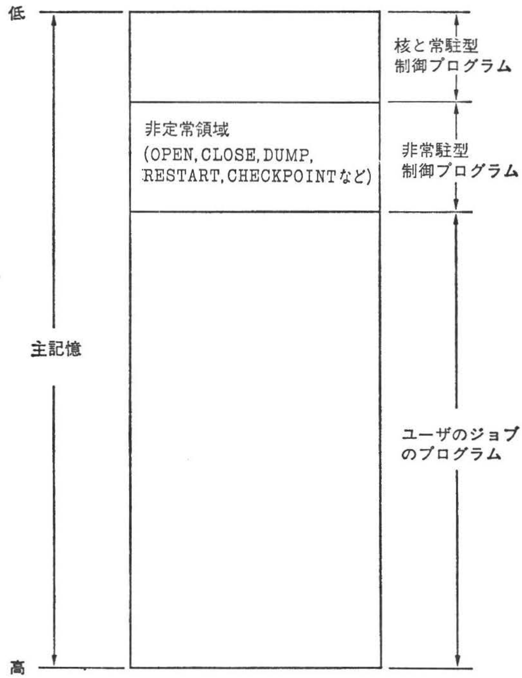


図 3.12　IBM 360/BOS の記憶の配置

PUT のような論理入出力パッケージと同様に，これらは，システムの一時的領域およびユーザの領域に格納される．(物理 IOCS と論理 IOCS の対比については，第2章 2.5.2.3 節を参照のこと)．

PIOCS は，物理入出力要求とその実行とを扱う入出力命令実行系の別名である．要求と実行とを監視することにより，PIOCS はチャネルの使用をスケジュールできる．したがって，PIOCS の別名を**チャネルスケジューラ**(channel scheduler)ともいう．

一つのユーザジョブから別のジョブへの遷移を監督するシステムプログラムを，**ジョブ制御**(job control)という．ジョブ制御は，一つのサブシステムから別のサブシステムへの遷移も監視する．IBM 360/BOS のサブシステムには，アセンブラ，レポート生成，整列/併合，プログラム実行フェーズ，虫取りシステムがある．サブシステムを使うことは，ジョブとして扱われるので，ジョブの遷移の監督とサブシステムの遷移の監視とは，全く同じ仕事になる．さらに，サブシステムの監視の機能は，ジョブ制御が行なう．たとえば，アセ

ンブラの監視において，ジョブ制御は，（ユーザのジョブ制御カードの指定に基づき）オブジェクトモジュールがシステムの再配置型ライブラリの中にあるのか，または，ロードモジュールがコアイメージのライブラリの中にあるのかを決める．ユーザが，そのジョブカードの中でアセンブルと実行のオプションを指定したならば，ジョブ制御は，アセンブラをロードして実行し，オブジェクトモジュールを再配置型ライブラリに挿入することをアセンブラに指示し，リンキングローダ（**リンケージエディタ**という）を呼び出し，ロードモジュールをコアイメージのライブラリに挿入することをリンキングローダに指示し，最後にロードモジュールを主記憶の中で実行する．ユーザの特殊な指定がなければ，ライブラリ内のロードおよびオブジェクトモジュールは，将来使用

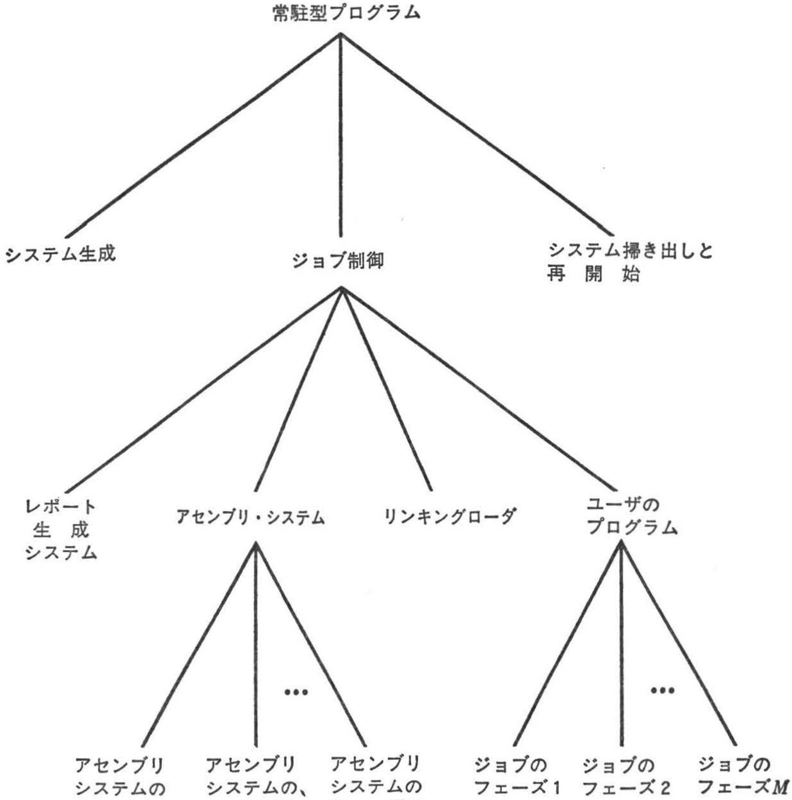


図 3.13 IBM 360/BOS のシステムの階層

するために退避することはない．一方，そのような指定があれば，これらのモジュールに対するカタログを作成し，ジョブ制御がそれを永久的に保持する．アセンブラとリンキングローダはかなり大きく，また，IBM 360/BOS は，8 K バイトの小さい主記憶と独立したディスクという構成に対して作られているので，これら2つの処理系のロードモジュールも，実行フェーズに編成してある．フェーズの名前は，勿論カタログの中にあり，フェーズは，コアイメージのライブラリの中にある．カタログとライブラリは，両方ともディスクに格納してある．同様にして，ユーザプログラムのロードモジュールも，フェーズに分けてある．したがって，システムローダを呼び出して，(言語)処理系やユーザプログラムの実行フェーズをロードするのは，ジョブ制御の機能である（図 3.13）．

この議論からわかるように，ジョブ制御は，一つの言語サブシステムから別の言語サブシステムに完全に自動的に遷移することを実現しているわけではない．いろいろの言語で書いたユーザプログラムは，異なったジョブに編成されており，中間結果は，一つのジョブから別のジョブに人手で渡すか後程使うために登録したファイルに特別にしまっておくので，ユーザとオペレータは，ジョブの準備を責任をもって行なう(図 3.14)．

一方，ジョブ制御は，言語処理とユーザプログラムの実行に対しては，良い自動的手続きを提供している．前者の場合，ジョブ制御は言語システムモニタとして働き，それがユーザプログラムの翻訳のために言語処理系を持ってきたり，翻訳したプログラムを（再配置型の）ライブラリに入れたり，リンキングローダを呼び出してプログラムフェーズの構成やコアイメージのライブラリへの挿入などをする．後者の場合，ジョブ制御は，リンキングローダを呼んでライブラリに対するプログラムフェーズを準備したり，プログラムフェーズをロードし実行したり，その実行の監視をするなどして，ユーザプログラムの実行の監督をする．それゆえ，これらの場合，ジョブ制御は，一般化した一括処理システムにおけるモニタとして機能する．IBM 360/BOS の詳しい説明については，読者は [10] および [11] を参照されたい．

3.2.3.2 ジョブ制御言語の利用

一括処理システムのコマンド言語あるいはジョブ制御言語は，オペレータとユーザとが，目的とするサービスのためのシステムの処理系および副処理系を動かしたり選択したりするために使う．これらのコマンドの構文は，システムごとに異なっているけれども，これらのコマンドを正しく使うためには，個々

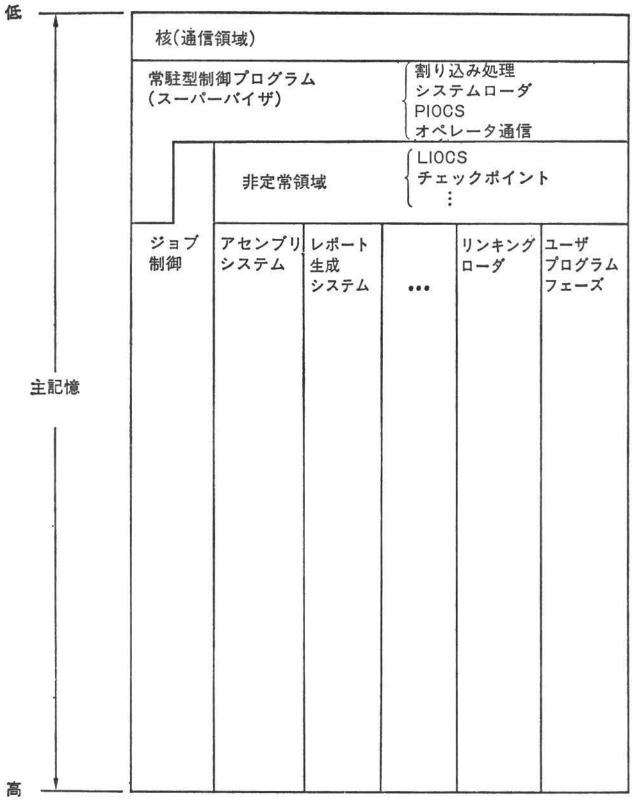


図 3.14 IBM 360/BOS に含まれるサブシステムの主記憶の論理的配置

のコマンドの構文の特性を詳細に気にかけるだけでなく，コマンドの列を注意深く用意することが必要である．この列は，処理系および副処理系をどのように階層に組み立て動かすかということと大きな係わりがある．

たとえば，IBM 360/BOS では //EXEC コマンドによりジョブ制御はプログラム実行の準備をするので，これをジョブデックの適切なところに置かなければならない．プログラムを準備するためには，ジョブ制御はリンキングローダをジョブのために使うかどうかを判定しなければならない．プログラムをライブラリ（永久に登録したコアイメージのライブラリか，一時的に入れた再配置型のライブラリ)から持ってくるか，あるいは，システムの入力機器から読み込むかどちらかなので，3つの可能性がある．これを，順番に説明しよう．

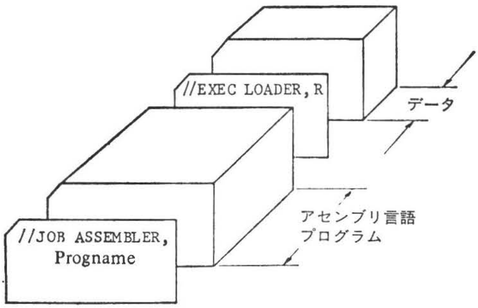


図 3.15　第 1 の 場 合

第1の場合　ユーザプログラムが，既にコアイメージのライブラリにある

//EXEC コマンドがあらわれたとき，ジョブ制御はリンキングローダを働かせない．そのかわり，ジョブ制御は，システムローダに命じて，//JOB カードに名前があるプログラムの第1プログラムフェーズをコアイメージのライブラリから持ってこさせ，また，ライブラリから持ってきたすべてのプログラムフェーズの実行を監督する．

後で実行するためにプログラムフェーズをコアイメージのライブラリに入れるために，//JOB SYSCMAINT コマンドを事前のプログラムフェーズのジョブで使わなければならない．

第2の場合　ユーザプログラムが，システムの入力機器にある．

この場合，ジョブ制御は，//EXEC ジョブ制御カードがあらわれたときに，リンキングローダを働かせる．リンキングローダは，//JOB カードに名前があ

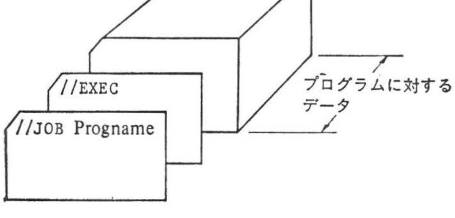


図 3.16　第2の場合

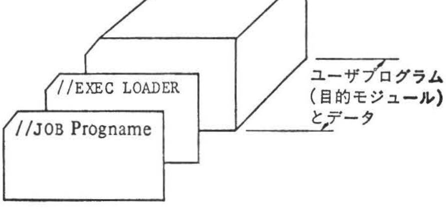


図 3.17　第3の場合

るプログラムをシステム入力機器から読み出し，プログラムが実行フェーズになる準備をし，プログラムフェーズの実行のためにジョブ制御に戻る．

第3の場合　ユーザプログラムが，再配置型のライブラリにある．

この場合，アセンブリ言語のプログラムをアセンブルしようとしている．プログラムの目的モジュールは，再配置型ライブラリに一時的に格納される．リンキングローダを，プログラムフェーズの準備のために持ってくる．リンキングローダは，目的モジュールの一時的な居場所を(//EXEC カードの文字 *R* によって)知る．そして，プログラムフェーズが実行される．

システムの階層から見ると，ユーザのプログラムフェーズの実行は，ジョブ制御によって，ジョブの準備と変換からジョブの処理に最後に遷移したことで始まると考えるのが合理的である．したがって，ジョブの最後に //EXEC を使うのは不便ではない．しかし，ユーザが //EXEC コマンドで指定して，自分のプログラムの居場所をジョブ制御に教える必要があるのは不便である．工夫してあるシステムでは，プログラムの居場所を決めるために，ライブラリ，システムの入力機器，ユーザの入力ファイルを組織的に探索する．これらの工夫は，ジョブ制御コマンドの使用におけるわずらわしさを軽減するけれども，システムの規模や複雑さを増してしまう．IBM 360/BOS のように記憶の少ないものに制限されているシステムにおいては，制御プログラムは小さくなければならない．それゆえ，ユーザはやはりジョブの準備の重荷を背負わされる．

3.2.4 オフラインの一括処理オペレーティング・システムにおけるいくつかの改良

ここまで述べてきた程度の一括処理システムでは，入出力と計算との重ね合せを充分行なうことは，困難である．この問題は，プログラムの命令を重ね合せのために編成することが殆んどない(あるいは，不可能なこともある)という事実に起因している．主記憶では，唯一つのプログラムが完了するまで実行されているので，CPU は，入出力操作が行なわれている間は，通常，暇である．一つの極端な場合は，オペレーティング・システムに対するジョブの入力をカード読み取り機を介して行なう**オンラインジョブ入出力**(on-line job I/O)という単純な場合である．ジョブ制御コマンドは，通常，それぞれカードに穿孔されるので，ジョブ制御カード，プログラムデック，およびデータを読んでいると，CPU は大部分の時間暇になる．同様の状況は，大量の出力情報をカード穿孔機に穿孔したり，印刷機に印刷するときにも生ずる．

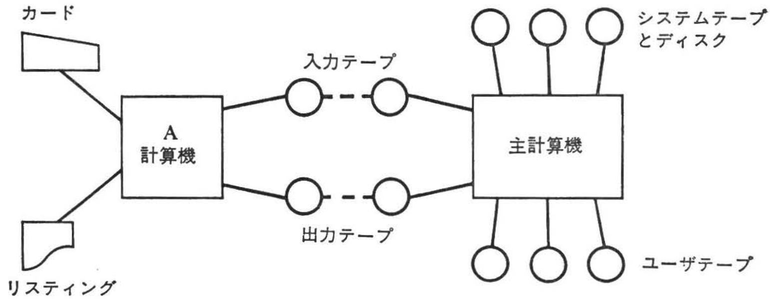


図 3.18　オフラインジョブ入出力

3.2.4.1　オフラインジョブ入出力

上述の状況をすり抜けるために，ジョブを一括化するために磁気テープを使う．主計算機が前のジョブの処理をしている間に，一つの計算機を使って一括化したジョブをカード読み取り機から読み，テープに書き出す．次に，そのテープを主記憶のテープ装置に装填し，主計算機が一括処理を始める．テープ装置は，カード読み取り機や穿孔機に比べて高速の入出力機器なので，ジョブの入力および出力のためにそれを使えば，CPU の空き時間を減少できる．

今までの議論からわかるように，カード読み取り機と入力ジョブテープの装置は，スーパーバイザがわかる適当なコマンドを使って切り換えることができる．したがって，ユーザのプログラムでカード読み取り機を参照している入出力命令は，今度はテープを実効的に参照することになる．印刷機やカード穿孔機とテープとを切り換えれば，一括化したジョブからの出力を保存のためにテープに退避することができる．テープは後で計算機にかけて，その印刷機にリストを出したり，そのカード穿孔機に穿孔カードを出したりする．

このように一括化したジョブを処理することを，**オフラインジョブ入出力** (off-line job I/O) という．

3.2.4.2　直接結合システム

オフラインジョブ入出力の機能をもった一つの特定のシステムは，Yale で実現されてきた [18]（図 3.19 参照）．このシステムでは，機械やオペレーティング・システムに，次のような修正を施してある．

1. 小型で低速の計算機がすべての入出力操作を処理し，主人として働く．大型で高速の計算機は計算だけを実行し，従者として働く．

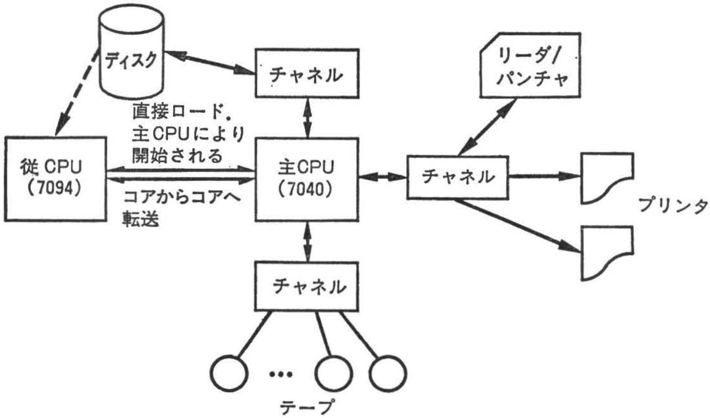


図 3.19　Yale 7094-7040 直接結合システム

2. 主計算機に，従計算機に対するロード，開始，および処理の監視という機能を与えるために，システムに，いくつかの機構を追加する．
   a) 各計算機が相手に割り込む機能．
   b) 主計算機の TRANSMIT 命令を修正して，2つの主記憶間でデータのブロックを移動できるようにする．
3. 主計算機のオペレーティング・システムが，従計算機の計算をスケジュールする機能．

項目 b) では，システムの2つの主記憶が直接結合されることが必要である．このために，このようなシステムを，**直接結合システム**(directly coupled system)という．これらのシステムでは，大規模な一括化ジョブをかなり容易に大量に処理することができる．

最近の直接結合システムの使用経験によれば，従計算機は空いているのに，負荷の大きい主計算機は入出力操作で沈没してしまう．したがって，OS/360-ASP や VS/370-JES2 のようなこの型の構成では，2つのうち大きい方を，主計算機にしている．さらに，主従の概念を少し変えて，従者は，(その平均的性能が計算限界になっている限り) 自分自身の入出力を選択でき，主人は，(その局所的な仕事の負荷が入出力処理を干渉しない限り)いくらかの局所的な処理を行なうことができる [12].

3.3 オンライン一括処理システム

オフラインジョブ入出力にしろオンラインジョブ入出力にしろ，どちらを使おうとも，オフライン一括処理システムでは，ユーザはそのジョブを穿孔したカードで提出しなければならない．たとえば，テレタイプライタを使うことは許されない．カードデックの形式でジョブを準備することは，任意にしてよい．テレタイプライタのような端末を排除してあるのが，不便である．

一括処理システムでオンラインプログラミングを行なわせようという試みがなされてきた．**オンラインプログラミング**(on-line programming)とは，ユーザが，穿孔したカードやテープのような中間記憶媒体を使わないで自分のジョブ制御コマンド，プログラムの文やデータを入力でき，オペレーティング・システムからその出力を直接受け取れることを指す．オンラインプログラミングの能力をもった一括処理システムを，**オンライン一括処理システム**(on-line batch processing system)という．

テレタイプライタや鍵盤付表示装置のようなオンラインの端末は，通常このような目的のために使う．しかし，その入出力の能力はカード読み取り機や穿孔機よりもずっと遅いので，CPU が空くという同様の問題が生じ，もっと悪くなる．オンラインプログラミングは多数の端末を使うことを目指しているので，この問題はさらに難しくなる．

テレタイプライタや鍵盤付表示装置をジョブの入力のために使うと，ジョブを穿孔カードとして準備する必要がなくなる．これは，ジョブのキーパンチに実質的にとって替る．しかし，同一のジョブを処理のために入力するごとに，もう一度テレタイプライタに全体のジョブを打ち込むのは馬鹿げている．当然，オンラインシステムは，ユーザのプログラム，データ，コマンドなどを後程使うために，退避しておくファイルシステムをもっていなければならない．

上の議論から，オフライン一括処理システムがオンライン一括処理システムになるためには，次のものを用意しなければならない．

1. テレタイプライタや鍵盤付き表示装置のようなオンラインの端末，
2. オンライン端末からの低速入力を処理する衛星計算機，
3. 一般的な計算をする中央計算機，
4. 集中型のファイル．

単純なオンライン一括処理システムは，2つの計算機，すなわち，データチ

ャネルや通信回線で結合した小型のものと大型のものからなる．小型計算機は，テレタイプライタや鍵盤付き表示装置のような端末をもっている．そのオペレーティング・システムは，ジョブをメッセージとして大型計算機へ送る．さらに，オペレーティング・システムは，大型計算機がジョブを開始し，変更し，終了させるように起動する制御機能を提供する．高速で大型の計算機の拡張した一括処理オペレーティング・システムは，複雑な計算をし，オンラインジョブおよびバックグラウンドジョブ(たとえば，オフラインジョブ入出力を介して入れたジョブ)に対してファイルを処理する．割り込みを介して，小型計算機は大型計算機と実時間で通信できる．

もっと工夫をしたオンライン一括処理システムは，一つの大型機と複数個の衛星計算機で構成する．たとえば，IBM 3791 通信制御装置は，その種の衛星計算機で，通信回線を介して大型計算機 IBM 360/370 につながっている．衛星計算機として，IBM 3791 通信制御装置は，バックアップやファイルのためにディスクやディスケットのような自分の 2 次記憶をもち，データバッファリングや処理のために主記憶をもち，命令の解読とプログラムの実行のために制御論理機構をもっている．これは，多数の鍵盤付き印刷機，鍵盤付き表示装置，および，通信用端末を処理できる [1]．

3.3.1 簡単なオンライン一括処理システム

ここで，簡単なオンライン一括処理システムについて，その機能的および構造的要素の観点から論じてみよう(図 3.20 参照)．

3.3.1.1 表，バッファおよびコマンド

A. バッファと新しいコマンド

オンライン端末からのジョブの入力と出力とは，文字を使って行なわれるので，システムはこれらの文字を集めるバッファを提供しなければならない．実際，3 つの型のバッファがある．

1. **衛星計算機のバッファ．** これらのバッファは，通常非常に小さい――たとえば，64 字である．バッファの大きさは，中央の計算機に転送する文字の数，端末の入出力率，衛星計算機で使える主記憶によって決められる．
2. **中央の計算機のバッファ．** これらのバッファは，衛星計算機のものに比べていくらか大きい．しかし，それは，中央の計算機の主記憶の大き

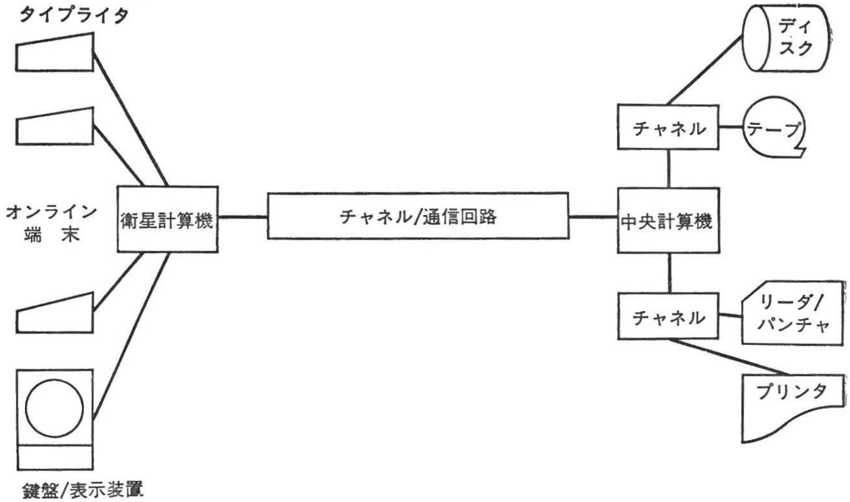


図 3.20　簡単なオンライン一括処理システム

さに比べれば，やはりまだ小さい――たとえば，500語である．これらのバッファの大きさは，衛星計算機に送り出す文字の数，ディスクなどの2次記憶機器の入出力率，中央の計算機で使える主記憶などによって決まる．

3. **オンラインバッファ**．これらのバッファは，ずっと大きい．これらは，システムによってディスクのような2次記憶機器上に作られる．これらのバッファの大きさは，数千語である．また，これらのバッファの大きさも，中央計算機に送り出す語の数，“平均的な”ジョブの大きさ，使用可能な2次記憶によって決まる．

衛星計算機内の入力バッファは，ユーザが文字（テレタイプから入力した場合）や行（表示装置コンソールから入力した場合）を入れるとすぐに文字や行を集めること，必要ならば文字コードをあるもの（たとえばASC II）から別のもの(たとえばEBCDIC)へ変換すること，伝送のために文字を詰めることなどを目的として作り出される．衛星計算機の出力バッファは，中央計算機からメッセージを受け取ること，メッセージの制御ビットを解読すること，必要ならば文字コードを変換すること，端末が出力できるだけの文字を伝送すること，および，端末を動かし続けるために使われる．

中央の計算機のバッファについては，工夫は凝らさない．衛星計算機および中央の計算機のバッファは，ユーザにとっては見えない．ジョブの入出力を扱うときの役割は，端末，衛星計算機，中央計算機および2次記憶機器間の速度，容量，機能における不釣合を埋めることである．しかし，オンラインバッファは，ユーザにとってはっきり見える．ユーザは，一括処理のために，自分のプログラムの文，データ，コマンドなどを蓄え，また，このバッファに自分の出力を格納することができる．このために，オンラインの端末と，オンラインの入力および出力バッファの組との間には，一対一の対応がある．さらに，オンライン一括処理システムは，ユーザに対し，新しいコマンドの集合を提供する．

1. 入力および出力バッファをきれいにすること．
2. 入力バッファに行くプログラムの文，データ，システムコマンドを追加すること．
3. 行を消すこと．
4. 行を挿入すること．
5. 行を編集すること．
6. 入力または出力バッファの行を印刷すること．
7. 入力バッファ内のジョブの処理を開始すること．

端末がオンラインの表示装置コンソールのときには，行の追加，削除，挿入，編集，印刷は，文字に関するものへ拡張できる．

便宜のために，オンラインバッファを単にバッファと呼ぶことにしよう．また，バッファ中の文，データ，コマンドを行と呼び，行番号を使って参照する．

B. 表とそれについてのコマンド

オンライン一括処理システムがユーザ，ジョブ，および端末を追跡してゆくために，**システム操作表**(system operation table)がある．この表には，次のような情報がある．

1. 端末の識別名．オンラインあるいはオフライン機器どちらにしても，すべての端末には，一意的な識別番号がある．
2. ユーザの識別名．
3. バッファの割り当て．各端末に対し，バッファの組，すなわち，入力バッファと出力バッファが与えられている．表のエントリは，はじめ，

記号装置名である.(記号装置名に関する議論は, IOCS に関する章の 2.5.2.1 節を参照してほしい). その後, エントリは, システム生成前にバッファが割り当てられた装置の装置制御ブロックのポインタに変る.

4. 操作状態. これは, 中央の計算機システムの現在の状態に関する情報である. システムは, この端末に対してジョブを実行しているか, あるいは, どこか他の端末で走っているジョブの完了を待っている. どちらの場合にも, ジョブが始まる行の行番号を, 表のエントリに退避しておく. ユーザジョブを実行していないときには, システムは, 行をきれいにすること, 行の追加, 削除, 編集または印刷を行なっている. そのような実行について, 表のエントリに記録しておく. そこでは行を扱っているので, 行番号がやはり必要である.

登録するためのコマンドを端末で出すと, 端末識別名およびユーザ識別名を検証し, 表に入れる. 同様に, 登録抹消のコマンドを出すと, ユーザ識別名を表から削除する.

3.3.1.2 衛星計算機のオペレーティング・システム: 少数の状態をもった有限状態オートマトン

衛星計算機のオペレーティング・システムの構成は, 次の要因によってきまってくる.

1. テレタイプライタとの通信は, 非同期式で文字ごとに行なわれる. これらの文字は, たとえば, 10 文字/秒の最大速度で入力される. 文字を出力するためには, テレタイプライタを最大速度に保つことが望まれる.
2. 表示装置コンソールとの通信は, 同期式で, 文字ごとに行ない, ある種のレジスタを介して行なう.
3. 中央の計算機との間の通信は, 握手(handshake)と呼ぶ形式をとる. 衛星側のオペレーティング・システムから始まるメッセージは, 中央の計算機のオペレーティング・システムからの返信によって確認しなければならない.

バッファをきれいにしたり, 行や文字を編集するようなオンラインジョブの準備の機能は, すべて中央の計算機で行なわれる. 衛星計算機オペレーティン

グ・システムの論理は，一般に，次の状態を含んでいる．

冬眠状態(dormant state)　テレタイプライタあるいは表示装置コンソールが，文字の入力あるいは出力を全然していない．

読み込み状態(read state)　1文字が(64文字分の)バッファに読み込まれる．

書き出し状態(write state)　1行分の文字を，機器に書き出す．印刷出力中止という特殊な信号によって割り込まれない限り，文字の行を印刷したり表示したりする．割り込まれた場合，緊急状態になる．

緊急状態(emergency state)　特殊な信号は，1文字を直ぐに読み込むことを指示する．そうしないと，その文字は無くなってしまうか，別の文字で上書きされてしまう．したがって，この状態では，システムは，即座に一時的な“時間切れ”信号を機器に送り，後程出力操作を再開するために出力すべき残りの文字数を表に記録しておく．そこまでくれば，次の状態は読み込み状態になる．

握手状態(handshake state)　文字の1行が，中央計算機のオペレーティング・システムに送られようとする．この文字の行は，握手が受け取られたときに，送り出される．そうでなければ，後でこの文字の行を出力するために，この状態に再び入ってくる．印刷出力中止という信号があらわれれば，割り込みが生じ得る．

処理状態(processing state)　コード変換と文字の詰め合せが，この状態の基本的処理の仕事である．しかし，IBM 3791 通信制御装置のようなもっと進んだ衛星計算機では，処理の仕事は，メッセージの編集，表示の形式化なども含んでいるだろう．これらの仕事をすれば，衛星計算機は，中央計算機がそのようなことをしないで済むように支援できる．

衛星計算機のオペレーティング・システムを有限状態オートマトンとして設計するのは，難しい仕事ではない．これは，演習問題として，読者に任せよう．

3.3.1.3 中央計算機のオペレーティング・システム

システム操作表と割り込み処理ルーチンをもつために，核と常駐型制御プログラムの拡張が必要になる．さらに，非常駐型の制御プログラムの数と大きさも大幅に拡大しなければならない．一つの重要な拡張は，ファイルシステムで，システムとユーザが，プログラム，データ，コマンド手続きなどを格納し更新できるようにする(**図3.21**参照)．

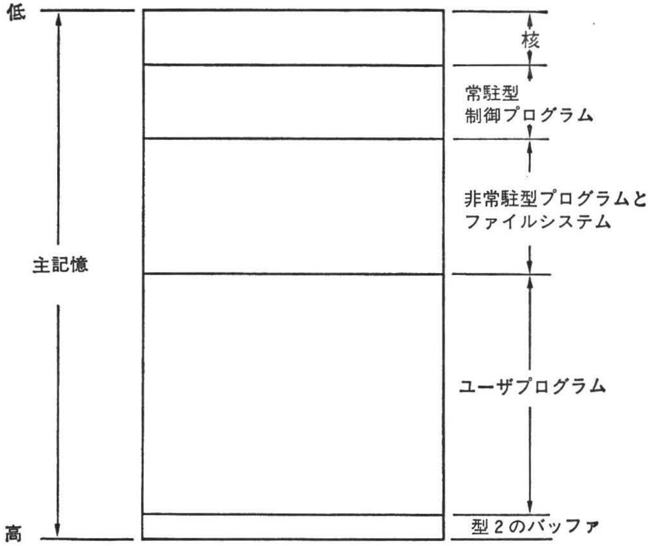


図 3.21　中央計算機のオペレーティング・システム

3.3.1.3.1　常駐型制御プログラムの役割の変更：割り込み処理系の研究

中央計算機のオペレーティング・システムと通信するために，衛星計算機のオペレーティング・システムは，データチャネルに要求を出し，それが中央の計算機に割り込みをかける．この割り込みは，常駐型制御プログラムの一つである**割り込み処理系**が処理をする．割り込みが生じたとき，システムは次の3つの状態のどれかである．

実行可能状態

システムは，入力バッファのある行の追加，削除，編集，挿入，処理の開始や，バッファをきれいにすることなどのオンラインプログラミングのサービスと無関係のプログラムを走らせている．この場合，割り込み処理系は，衛星計算機のオペレーティング・システムからの要求，割り込まれたプログラムが使っていたすべてのレジスタの現在の内容，割り込まれたプログラムを後で継続させるためのプログラムカウンタの内容を退避する．さらに，割り込み処理系は，要求のパリティを検査し，必要な入出力操作がすべて完了するまで待たせ，また，要求の解読とサービスをする．要求したサービスが完了した(たとえば，ある入力バッファから1行を削除した)ときには，割り込み処理系は，

すべてのレジスタの内容を回復し割り込まれたプログラムを再開する.

使用状態

この状態では，割り込み処理系は，要求のサービスをしている最中である．現在の要求が完全に処理し終らないうちに新しい要求を受け付けることはしないので，割り込み処理系は，メッセージを衛星計算機のオペレーティング・システムに送り，後で要求を再出させるようにする．そのうちに，割り込まれたプログラムは，実行を再開する．

待ち状態

この状態では，割り込み処理系は，サービスルーチンを続けるために，衛星計算機からもっとデータを受け取ることを待っている．

サービスが中断できなければ，使用状態になる．サービスのためにデータが必要なときは，待ち状態になったまま，システムはそれを待つ．したがって，割り込み処理系は，オンラインプログラミングのサービスに常に優先権を与えている．

3.3.1.3.2 拡張した非常駐型制御プログラム

非常駐型制御プログラムの主な拡張は，ファイルシステムを追加することである．ファイルシステムをオンラインで使用することには，次の2つの場合がある．

1. ジョブの準備の中で，ファイルシステムを使うこと．
2. 実行時にデータやプログラムにアクセスするためにファイルシステムを使うこと．

2番目のオンライン的なファイルの使い方については，この章では述べない．第1のオンライン的なファイルの使い方に関しては，ジョブの準備のためのコマンドに関心がある．前にも述べたように，入力バッファ中のジョブの行をきれいにし，追加し，削除し，編集し，挿入し，印刷するためのコマンドはある．しかし，ジョブの一部を構成しなければならないデータ，プログラムやコマンド手続きをファイルの中で指定するコマンドは用意してなかった．このために，そのような目的に対する新しいコマンドを準備しなければならない．

それゆえ，衛星側と中央計算機との接合部をなす非常駐型制御プログラム(ファイルシステムを除く)は，次の3つの主要な機能をもっている．

1. 中央計算機のオペレーティング・システムのための入力バッファ中に

あるジョブの入力の前処理.

2. 中央計算機のオペレーティング・システムからのジョブの出力の後処理.
3. オンラインコマンドの解釈.

A. 入力前処理

入力バッファ中にあるジョブの前処理を必要とするのは，次の3つの理由による.

1. ジョブの文を走査して，オンラインコマンドを解釈しなければならない.
2. 入力バッファ中のジョブの文は，中央計算機のオペレーティング・システムが処理するのに適切な形式になっていないかもしれない.
3. ユーザが与えたコマンド以外に，いくつかの追加すべきコマンドを制御プログラムが挿入して，ジョブに対するシステム出力の後処理をさせる.

ジョブの文を前処理することにより，システムは，ユーザにもっと自由なコマンド言語の構文を使えるようにする．たとえば，自由な構文の行からカードイメージへの変換は，この前処理系で行なう.

プログラムやデータの集合を前処理のためにバッファから主記憶へ転送しなければならないとき，システムは一括処理システムなので，ユーザは，この集合が一括化したジョブを構成しなければならないということはわからない．このため，ジョブ終了のコマンドおよび他の必要なコマンドを，前処理系が与えなければならない．さらに，実行しているジョブがシステムの入出力装置を使うための標準的手続きも，避けなければならない．ユーザが特に要求しない限り，入出力バッファの存在する装置を使わなければならない．装置切り換えのコマンドは，前処理の段階で挿入される.

B. 出力後処理

ジョブの出力の後処理が必要となる条件が2つある.

1. 正常に穿孔される2進出力が，あとで使うためにファイル内に格納してある.
2. 正常に印刷される英数字出力が，ユーザの出力バッファに入っている．入力の場合と同じように，形式上の違いがあるので，さらに変換ス

テップが必要になる.

C. オンラインコマンド言語の解釈

ユーザは，自分のジョブの一部として事前にファイルに格納してある原始あるいは2進の情報を使うことが許されているので，システムでは，ジョブに必要な情報が，すべてユーザの入力バッファに実際に入っている必要はない．たとえば，端末から，バッファ内に2進のデックを取り入れることはできない．したがって，2進デックは，ファイルの中に格納し，ファイルシステムによって，バッファの中に読み出してこなければならない．格納した情報を検索するためには，システムは，適当なジョブ制御カードを生成し，バッファ中のユーザのジョブに挿入する．

D. 偏りをもつスーパーバイザと永久的スーパーバイザ

D 1. ジョブスケジューラ

ジョブスケジューラは，次に走るべきジョブを実際に決めるスーパーバイザの部分である．オフラインの一括処理システムでは，ジョブスケジューラは，常にバッチの中の次のジョブを開始させるために，下に述べる方法をとる(3.2.2.1節参照).

1. カード(あるいはカードイメージ)を読むこと.
2. カードがジョブの開始を指定しているかどうかを決定すること.
3. そうであれば，料金計算ルーチンを働かせること.

オンライン一括処理システムでは，ジョブスケジューラは，常に端末に優先権を与えているが，次のステップを踏む.

1. カード(またはカードイメージ)を読み込む.
2. カードがジョブの開始を指定しているかどうかを決定する.
3. カードがジョブの開始であれば，ステップ5.につづく.
4. 割り込まれたジョブを再開し，残りのステップを飛ばす.
5. そのジョブがオンラインジョブがどうかを決定する.
6. それがオンラインジョブでなければ，オンラインジョブをさがす.
7. もしあれば，オンラインジョブに対する料金計算ルーチンを働かせ，もしなければ，バックグラウンドのジョブに対するものを働かせる.
8. 新しいジョブを続ける.

D 2. オンラインジョブの優先度制御系

オンラインジョブは，バックグラウンドジョブに対して優先度をもっているだけでなく，オンラインジョブの中にも，優先度をつける可能性もある．優先度の割り当ては，任意でよい．しかし，優先度制御系を実現して，それが，3.3.1.1節に述べたシステム操作表のエントリの順序に従って，次に処理すべきジョブのスケジュールを常に行なう．

D 3. 永久的スーパーバイザ

端末のユーザにはわからないが，オフラインとオンラインの一括処理システムでのジョブの概念は，全く異なっている．オフラインの環境では，ユーザは，ジョブの開始とジョブの終了のカードの間に自分のプログラムとデータを置いて，ジョブを提出する．しかし，オンラインの環境では，ユーザは，登録および登録抹消を唯一回する積りであり，その間に多数のプログラムの仕事を入力し，異なった時点でいろいろの結果を待っている．したがって，ユーザに関する限り，ジョブという概念は，登録および登録抹消の間にあるプログラムの仕事の会話的セッションに相当する．しかし，実際，システムは，オフライン一括処理システムの意味で，(この行から開始するというコマンドにより起動される)個々の仕事をジョブとして扱う．3.2.1.4節で述べたように，ジョブ間の遷移は，スーパーバイザが扱う．オーバーヘッドを減らし，ジョブの遷移を実現するために，スーパーバイザは，インターバルタイマを(自動的に)セットし，端末のユーザに対し同一の入出力構成を保持する．このため，スーパーバイザの一部は，永久的に主記憶に存在する．

3.3.1.4 ファイルシステム

ファイルシステムは，オンライン一括処理システムにとって本質的であるばかりでなく，後の章で述べるシステムの主要な部分ともなっている．このため，ファイルシステムに関する議論は，この教科書の後半部まで延ばすことにしよう．

3.3.2 オンライン一括処理オペレーティング・システムの要約といくつかの改良

オンライン一括処理システムは，限られた数の端末の下にある小さなオンラインプログラミングのジョブにとって，役に立つものである．座標に関してかなりの計算が必要となるグラフィック表示装置は，速く結果を出すために，中央の計算機へ送ることができる．オンラインプログラミングの情報——プログ

ラム，データやジョブ制御コマンド――は，後で検索や更新をするためにファイルに退避することができる．衛星側も中央側も，単独では多重処理の機能(すなわち，入出力と計算の重ね合せ以外にCPUによる並列的計算)をもってはいないけれども，結合したシステムは，実際には，2つのCPUをもつ多重処理システムになる．

各ユーザがシステムに登録すると，中央計算機のオペレーティング・システムは，ユーザに対し2つのバッファ――入力バッファと出力バッファ――をディスク上に作成する．入力バッファは，ユーザのコマンド，プログラムやデータをスタックするために使い，それらを，システムの提供した入力や編集の機能を使って，編集したり，調べたりする．入力が中央の計算機で処理できるようになると，ユーザはオンラインコマンドを発する．このコマンドにより，中央計算機は，ユーザの入力バッファにあるジョブの処理をスケジュールする．ジョブの処理では，次のことを行なう．

1. コマンド，プログラムやデータをユーザのファイルのレコードとして格納すること．
2. 公共およびユーザの私用のファイルから，コマンド，プログラムやデータを検索すること．
3. プログラムや手続きのアセンブリ，コンパイルや実行をすること．
4. ファイルとレコードの作成，更新および削除をすること．

ジョブの出力は，ユーザの出力バッファにためてある．ユーザの要求により，システムのオンライン出力コマンドを使って出力バッファを調べそれを印刷する．

3.3.2.1　間接結合システム

オンラインシステムの基本的システムの概念は，共用するディスクバッファやファイルを介して，入出力用計算機と大規模な計算用の計算機とを結合することである．2つの計算機の主記憶装置は直接結合しておらず，各計算機は，自分自身の仕事のスケジュールをしている．このため，この型の結合システムを，**間接結合システム**(indirectly coupled system)または，**ファイル共用システム**(shared file system)と呼ぶ．

高度なファイルシステムの機能をもった間接結合システムは，ペンシルバニア大学で実現されてきた(図3.22参照)．もっと工夫を凝らしたシステム構成も，間接結合システムで考えられている [2]．シミュレーションによる研究の

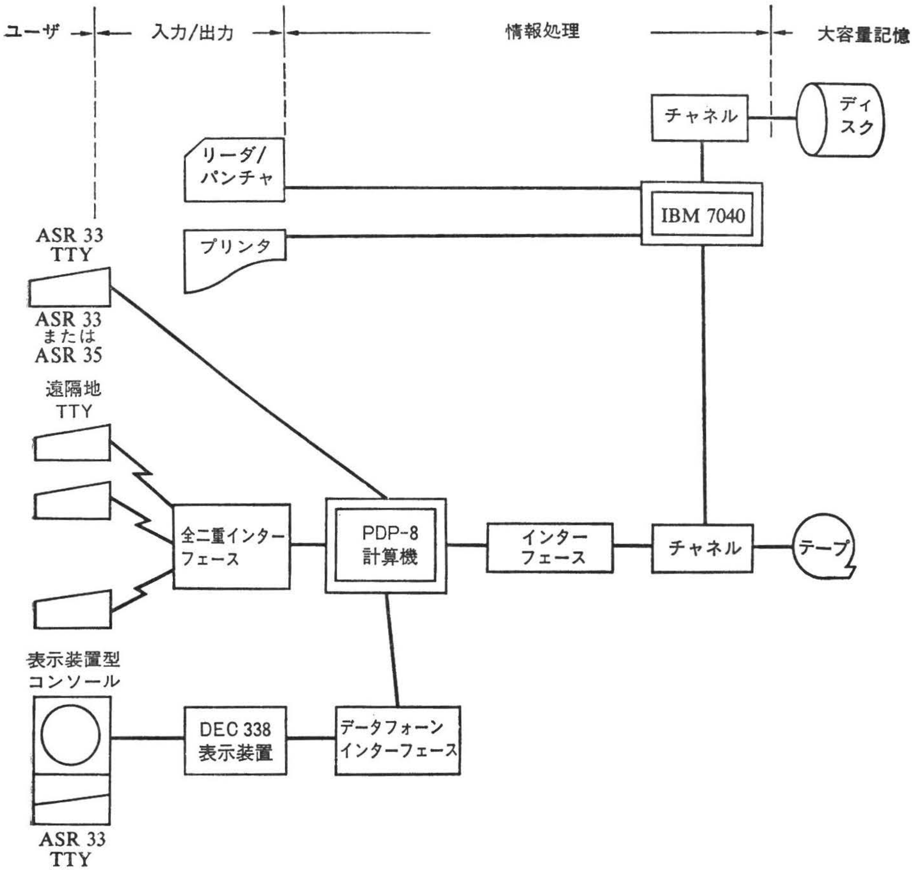


図 3.22 ペンシルバニア大学 Moore 校の結合システム/PDP 8-IBM 7040

結果，ジョブの**一巡時間**(turn around time)――すなわち，ジョブを提出してから戻ってくる迄の経過時間――およびジョブの処理量――すなわち，与えられた資源の集合の下で実行できる仕事の量――に関して，単独の一括処理システムよりも結合システムの方がずっと改良されることがわかる．

3.3.2.2 遠隔ジョブ投入

間接結合システムにおいて，たとえば，高速の読み取り機/印刷機の端末を介して，通信回線により遠隔地に置かれたステーションからジョブのデックを入力することが流行している．これらの端末を，**遠隔ジョブ投入**(remote job

entry)の端末という．ジョブデックをテープやディスクパックに写して中央計算機へ処理のために運ぶオフラインジョブ入出力(3.2.4.1節)とは異なり，遠隔ジョブ投入の端末にあるジョブデックを，通信回線を介して直接中央計算機に入力する．それゆえ，遠隔ジョブ投入は，オンラインジョブ入出力のずっと進んだ形式であり，その高速性，膨大性および遠距離性が特徴である．遠隔ジョブ投入の機能をもった間接結合システムは，オンラインシステムとも呼べる．しかしながら，前の議論をみると，オンラインジョブ入出力とオンラインプログラミングの間には，重要な違いのあることに気がつく．読者は，この違いについて，良く知らなければならない．ここでは，オンライン処理システムとは，それがオンラインあるいはオフラインのジョブ入出力のどちらを使うにしろ，オンラインプログラミングの能力をもったシステムのことを指す．

3.3.2.3 システムの構成

今までの議論からわかるように，オンラインシステムでは，衛星計算機が端末の入出力を，中央計算機が一括処理をする必要があった．勿論，そのような構成は，機能的に見たものである．計算機のハードウェアとソフトウェアを物理的に構成する場合，同じ機能的必要性は，いろいろの方法で達成できる．ある機能的要素をハードウェアで実現し，残りの機能的要素をソフトウェアで実現することを決定するのは複雑な問題であるので，そこから，計算機のハードウェアとソフトウェアの構成の幅広さが生じてくる．CDC6600 [21] は，恐らく，オンラインシステムの機能的要素が主としてハードウェアで実現されているという一つの極端な形態の一つであろう．このシステムは，1台の中央処理装置と10台の周辺処理装置とから成る．各処理装置は，自分自身の主記憶をもっている．周辺処理装置の記憶は，10個のチャネルを介して中央処理装置の主記憶につながっている．システムの制御は，常にこれら周辺処理装置のうちの1台に存在する．さらに，周辺処理装置は，他の処理装置のことは無視して，チャネルの動作の監視と，周辺記憶と中央の記憶との間のデータの伝送を独立に実行できる．中央処理装置は，浮動小数点演算を必要とするような計算限界のプログラムを処理するように設計してある．周辺処理装置は，大量の入出力操作を伴なうような小型の仕事を考えて設計してある．したがって，プログラムの準備やファイルの編集などのような仕事は，大部分周辺処理装置で実行する(図3.23参照)．

もう一方の極端な例は，機能的要素を主としてソフトウェアで実現しているオンラインシステムである．物理的には，周辺および中央の処理は同じCPU

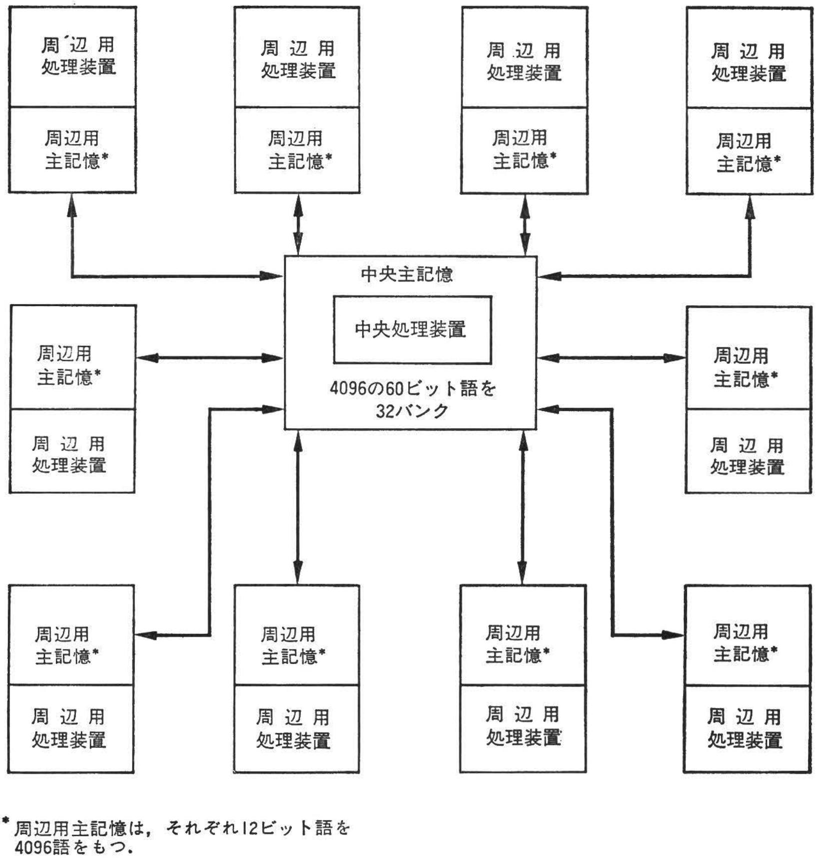


図 3.23　Control Data 6600 処理装置構成

が行なう．このようなシステムの一つの例に，IBM OS 360/370 のHASP [19] がある． このようなシステムにおいてさえ，OS 360/370 のような中央処理システムの役割と， HASP のような周辺処理システムの役割とは， 明確に区別している．

次節では，周辺処理システムの機能的役割と，周辺システムと中央のものとの間の関係に注意を集中して述べよう．

3.3.2.3.1　ジョブスプーリング・システム

3.2.4.1　節では，オフライン一括処理システムに対するジョブ単位の扱い

をする改良した方法，すなわちオフラインジョブ入出力として知られているものを述べた．その中では，オフラインジョブ入出力を実現する中間の機器は，テープであった．オンライン一括処理システムに対しても，改良したジョブの扱いをしてみたい．オフラインジョブ入出力の場合のように，改良したジョブの扱いに対する要求は，次の事項から推し進められる．

1. 機器やチャネルの割り込みを減らし，機器やチャネルの処理量を増加させたいこと．

   入出力操作に関する章で学んだように，各入出力操作は，チャネルコマンドと機器オーダの列で構成されていた．入出力実行系（IOCS の一部）による入出力操作の実行では，主記憶の中から適当なチャネルコマンドと機器オーダの列を取り出し，コマンドをチャネルにオーダを機器制御装置へ転送する．これらの機器（端末）に対するカードを読み込む（行を印刷する）というような入出力命令は，1枚のジョブカード（1行）の入力（出力）だけを引き起こすことができる．ジョブのバッチを処理するために，これらの命令を頻繁に実行しなければならない．しかし，入出力命令を頻繁に実行すると，読み込み終了信号のような割り込みが生ずる．割り込みの数を減らす一つの方法は，入出力命令をさらに長いチャネルコマンドと機器オーダの列に“つなぐ”ことである．1枚のカード（1行）を割り込みの間に読み込むかわりに，一連のカード（行）を割り込みが生ずる前あるいは停止が生ずる前に読み込むことができる．その結果，カード読み取り機/穿孔機や端末は，その最大速度で働く．

2. 特殊なブロッキング，デブロッキング，および，バッファ割り当ての機能を提供したいこと．

   IOBS（IOCS の一部）は，一般的なバッファリングの機能をもっているけれども，ジョブ入出力に対する特殊なバッファリングの機能を必要とすることがある．端末と単位レコード機器は，固定長のカードと行イメージを扱うので，バッファの大きさは固定している．それゆえ，ジョブ入出力に対するもっと効率の良いバッファ管理ができる．さらに，端末と機器の数もわかっている．したがって，必要なバッファの数は，使用可能な主記憶に基づいて決まる．バッファの切り換えとプール化とは，バッファが同じ大きさなのですぐに出来る．そのため，特殊なバッファリングの管理を使えば，単位レコード機器および端末へまたはそこからのジョブのバッチの流れを割り込みや停止を少なくして維持するこ

とが可能となる．

3. ジョブの優先度とジョブの多重投入の管理をしたいこと．

ジョブは，処理したいユーザのプログラムとデータの論理的な集まりをなしているけれども，それは多くの部分から構成されている．ユーザがジョブの異なった部分を異なった端末や機器からオンライン一括処理システムに投入できれば，便利であろう．ジョブの部分を混ぜ合わせることに関する適当なジョブ制御カードとコマンドが含まれている限り，部分を集めて単一のジョブとして準備することが可能でなければならない．収集と準備の過程では，オペレータやユーザが外部から割り当てたジョブの優先度を実現するために，ジョブ処理の待ち行列に並べる．

4. ジョブの前処理および後処理をしたいこと．

ジョブは処理される前に端末や機器から読み込まねばならないので，ジョブ(またはジョブカード)の前処理は，簡単にできる．前処理においては，あるジョブは，たとえば大きさ超過とか時間推定が大きすぎることにより，以後の処理を飛ばしたり削除したりできる．ジョブによって生成される出力は，同じ理由で，後処理により抑えることができる．前処理では，ファイルシステムとライブラリに入っているデータとプログラムを，要求しているジョブのために取り出してくることができる．後処理では，ジョブからの永久的な出力を別のジョブとして提出しないでも，直接ファイルシステムやライブラリに格納することができる．

要約すると，改良したジョブの処理は，中央処理システムに対する割り込みを減らすこと，周辺処理システムに専有の機器やチャネルの処理量を増やすこと，周辺処理システム専有のいろいろの機器や端末からジョブのバッチの流れを一定に保つこと，ジョブのバッチの前処理や後処理をすることなどを目的としている．

ここで，ディスクを中間的記憶として使う周辺処理システム，すなわち，Houston Automatic Spooling Priority System(HASP) [19] で使ういくつかのアルゴリズムと設計の考え方について調べてみよう．

基本的には，HASP は次のような仕事をする．

1. すべてのシステムの入力端末および単位レコード機器から，すべてのユーザのジョブ(すなわち，ジョブカードまたはジョブ文)を読む．
2. すべての入力してきたジョブを，中間的な大量のディスクに格納する．

3. ジョブの前処理をする.
4. 中央処理システムと交信して，それが前処理し格納したジョブの処理を始める.
5. 実行しているジョブが生成したすべての出力を蓄える.
6. ジョブの後処理をする.

HASPは，入出力命令の連結によってチャネルや機器の割り込みを減らしチャネルや機器の処理量をふやしたり，特殊なバッファリング機能を提供したり，ジョブの優先度を管理したり，ジョブバッチの多重投入を許したりして，1番目の仕事を能率よく行なう.

2.と5.の仕事を能率よく行なうために，HASPはディスクのモジュールから成る中間記憶の管理を能率良くできなければならない．ディスクの可動アクセス腕（機構）の並列性を利用し，ディスクの空間の追跡をするために，HASPは，いくつかの表とアルゴリズムとを使っている.

I. 表

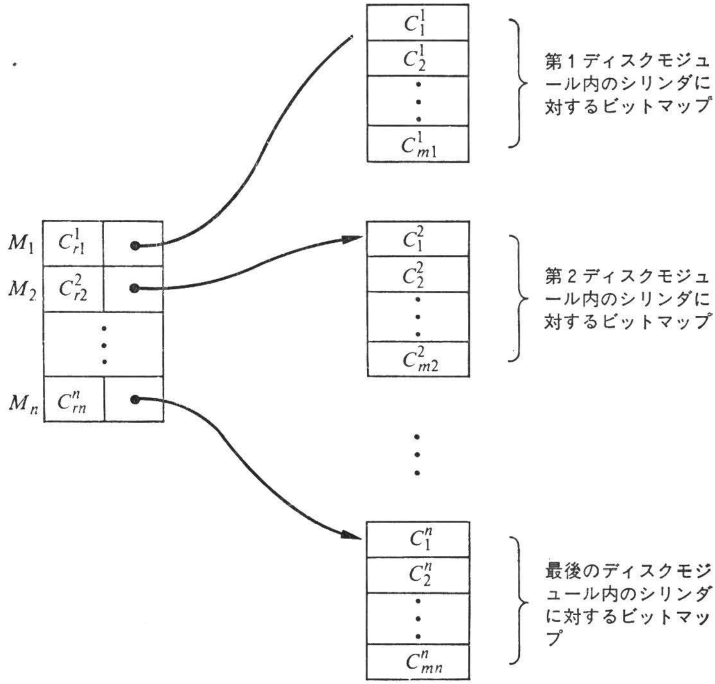
A. 主シリンダマップ

HASPには，主シリンダマップしかない(図3.24参照)．一つのシリンダのどのトラックもアクセス時間は同じで，現在のシリンダから別のシリンダへのアクセス時間はかなり大きいので，アクセスの能率のために考えるディスク記憶の最小単位は，シリンダである.

主シリンダマップは，2進ビットのマップで，マップ中のビットとディスクモジュールのシリンダとの間に一対一の対応がある．最初，すべてのビットは1で，すべてのシリンダは割り当てに使えることを示している．シリンダが割り当てられた直後に，ビットは0に設定される.

B. ジョブシリンダマップ

各ジョブに対し，HASPはジョブシリンダマップの対——一つはジョブとその入力データに対し，もう一つはジョブの出力データに対し——を作成する．本質的には，ジョブに対するジョブシリンダマップは，そのジョブとそれの入力および出力データにシリンダを割り当ててできる主シリンダマップの写しである．さらに，シリンダ内で使った最後のトラックの番地は，ジョブシリンダマップに覚えてあるが，これも記録しておく．このトラックの番地により，ジョブの入出力に割り当てるべき次のトラックを決定することができる．2つのシリンダマップがあるので，最後のトラックの番地が2つある.



M_i　　i番目のディスクモジュール

C_j^i　　i番目のモジュールのj番目のシリンダ

C_{ri}^i　　i番目のモジュール内で最も最近に参照されたシリンダ（すなわちri番目のシリンダ）

1. すべてのディスクが同一の型であれば，すべて同じ数のシリンダをもつ．この場合，$m1=m2=\cdots=mn$である．しかし，HASPは，異なった型のディスクも処理できる．
2. 各iに対するC_{ri}^iは，動的に更新される．

図 3.24　主シリンダマップ

C.　ジョブ制御表

各ジョブに対して，ジョブ制御表がある．この表は，ジョブについての料金計算の情報，ジョブカードにあらわれたデータ，シリンダマップへのポインタとそれに付随した最後のトラックの番地，それ以外の余分な空白からできてい

る．すなわち，ジョブ制御表は，シリンダマップへのディレクトリとして働く．

D. ジョブ待ち行列要素

各ジョブは，ジョブ待ち行列要素をもち，そこには，ジョブの内部的識別子，ジョブの外部的優先度の割り当て，制御表の場所，待ち行列の次の要素の番地の場所などが入っている．

II. 記憶(すなわちシリンダとトラック)割り当て手続き

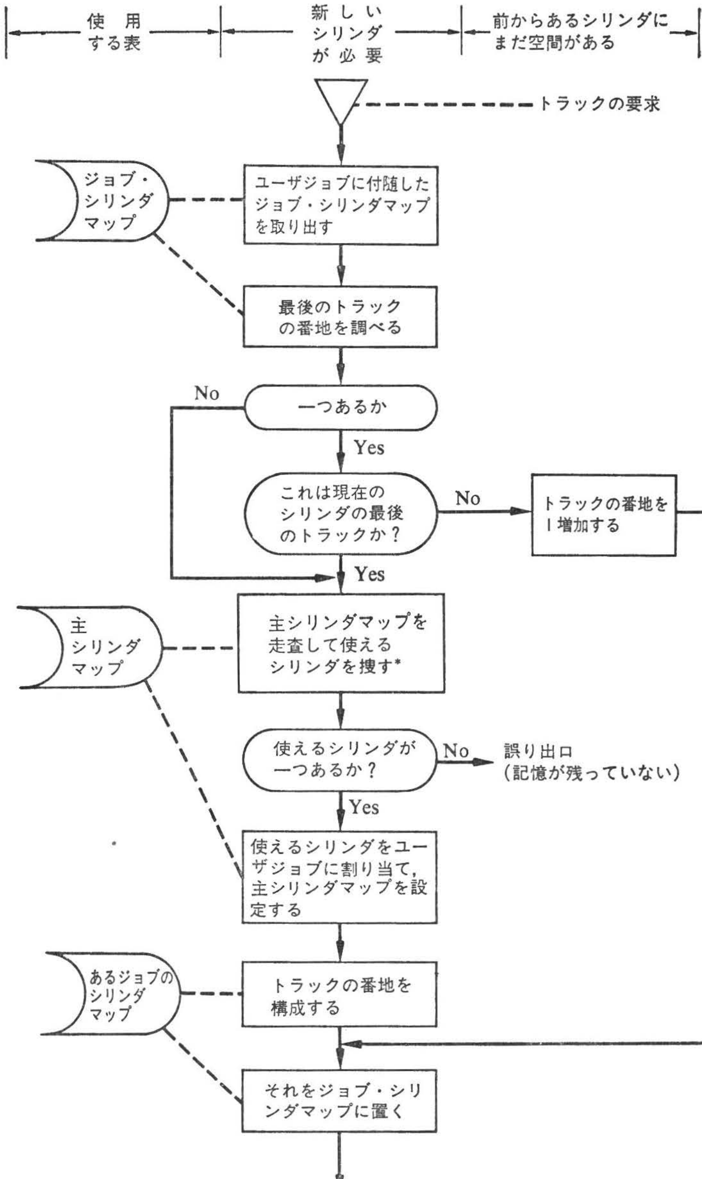
A. 記憶割り当て手続きは，専用のディスクモジュールを有効に使用するように設計されている．効率は，次のような方式により向上している．

1. シリンダは，アクセス腕(機構)の動的な位置に基づいて割り当てられる．したがって，アクセス腕の再位置づけは，最小に保てる．
2. ディスクモジュールのアクセス腕は，お互に独立に動くので，記憶の複数個のモジュールを同時に使うことができる．それゆえ，単一のファイルを，異なったモジュールの多数のシリンダへばらまくことができる．
3. シリンダは，アクセスを考えるときの最小の単位であるけれども，シリンダ内のトラックは，注意深く割り当てる．最初に，各トラックは，多数のカードイメージから構成される．カードイメージは，長さが固定で大きさも小さく，すべてのジョブはジョブカードとして扱うので，トラックの記憶には，断片(fragmentation)は非常に少ない．さらに，ジョブのすべてのジョブカードは，シリンダ内の連続したトラック上にある．

記憶割り当てのために，2つのマップを参照する．入力(出力)ジョブを格納するために割り当てるならば，主シリンダマップと入力(出力)ジョブシリンダマップを使う．主シリンダマップに反映されているように，使えるシリンダを探すために，工夫した探索アルゴリズムを使う．このアルゴリズムについては，もっと細かく調べよう．ここで，記憶割り当てを次の流れ図(**図 3.25**)で示そう．

III. シリンダ探索アルゴリズム

A. シリンダ探索アルゴリズムで使う唯一の表は，図 3.24 に示した主シリン



* 探索アルゴリズムを参照せよ

図 3.25　記憶割り当て手続き

ダマップである．探索には，3つの段階がある．最初の**直接探索**(immediate search)は，どこか使えるモジュールでもっと最近参照されたシリンダがあるかどうか決める．この決定は，主シリンダマップの各 i に対し，$C^i{}_{ri}$ エントリを調べればできる．もし使えるシリンダがあれば，ある j に対し，$C^j{}_{rj}$ はビットが1であり，j 番目のモジュールで1番最近参照されたシリンダが割り当てに使えることに示している．n 個のモジュールがあるので，n 個の1番最近参照されたシリンダがある．したがって，直接探索は，主シリンダマップの中で高々 n 個の $C^i{}_{ri}$ のエントリを調べればよい．このチェックは，速くできる．さらに，1番最近参照したシリンダを割り当てに使えば，ディスクのアクセス腕を再位置づけする必要もない．1番最近参照したシリンダで使えるものがなければ，そこで，2番目の段階，**近隣探索**(neighborhood search)に入る．近隣探索では，1番最近参照された一つのシリンダに隣接したシリンダ(両側で8個)を探す．割り当てに使えるシリンダがなければ，1番最近参照されたすべてのシリンダの隣接シリンダを探索する．換言すれば，モジュール M_i の $C^i{}_{ri}$ に対し，$r_i=k$ のとき，探索アルゴリズムは，

$C^i{}_{k-1}$ か $C^i{}_{k+1}$

のどちらかが使えるかどうか調べる．もし使えなければ，

$C^i{}_{k-2}$ か $C^i{}_{k+2}$

のどちらかを調べ，これを続ける．もし，

$C^i{}_{k-8}$ か $C^i{}_{k+8}$

のどちらも使えなければ，この探索アルゴリズムは，$(i+1)$番目のモジュール M_{i+1} についてチェックを繰り返す．

近隣探索は，ディスクのアクセス腕が近隣のシリンダへ到達するためのアクセス時間は，任意のシリンダへのアクセス時間よりも少ないという理論に基づいている．近隣探索が駄目なときは，第3段階の探索が始まる．この段階では，**全域探索**(exhaustive search)を使う．全域探索は，第1ディスクモジュール M_1 に対し，$C_1{}^1, C_2{}^1, \cdots, C^1{}_{m1}$ を調べることから始まり，続いて，次のディスクモジュールに対して調べ，これを，使用可能なシリンダ $C_j{}^i$ が見つかるまで繰り返す．そうでなければ，使えるシリンダは全然ないことになる．

一般に，このアルゴリズムの設計は，直接探索を頻繁に，近隣探索は時々，

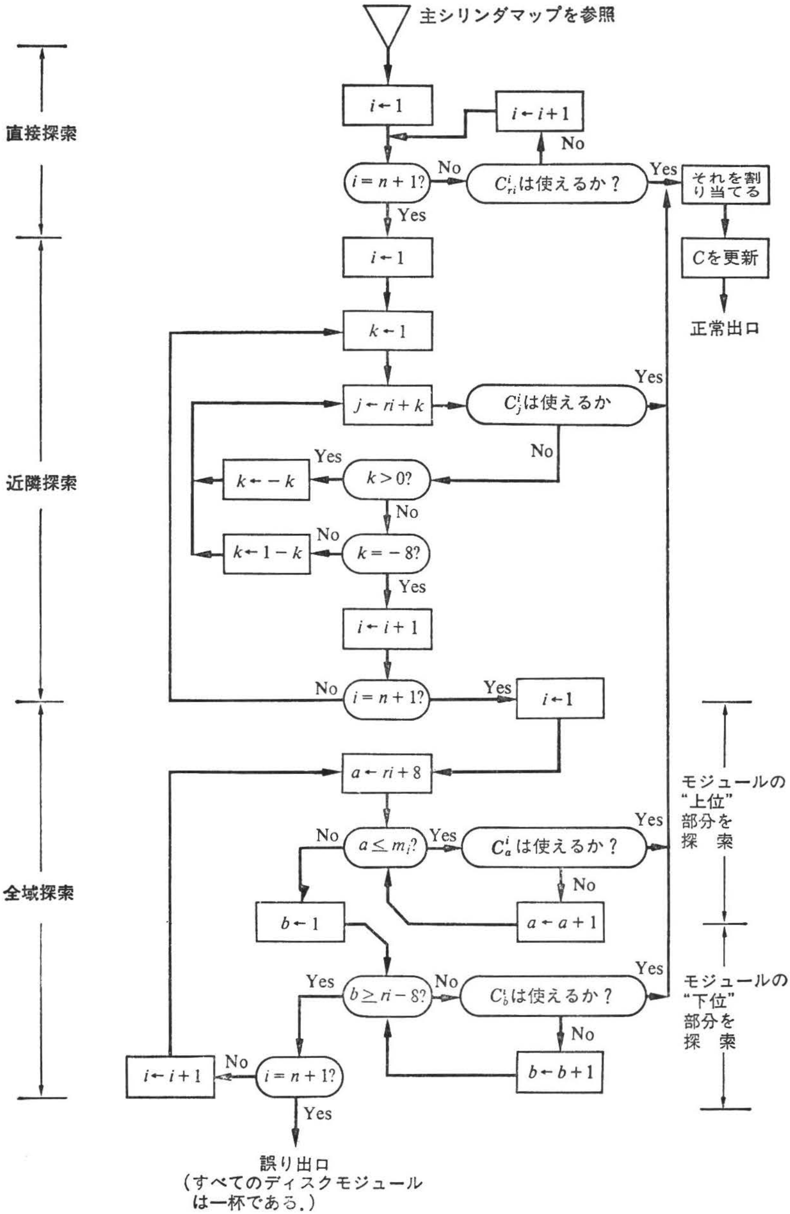


図 3.26　シリンダ探索アルゴリズム

全域探索は滅多にしないようにしてある．シリンダ探索アルゴリズムを，図 3.26 に示す．

仕事 3．と 6．(すなわち，ジョブの前および後処理)を行なうために，HASP は，ジョブ制御表とジョブ待ち行列要素とを利用する．ジョブの優先度は，待ち行列の形を決める．明らかに，待ち行列では同一の優先度のジョブの要素を結合してあり，高い優先度のジョブは，低い優先度の要素よりも前に結合した要素を並べておく．許可する処理時間と出力の量は，ジョブ制御表によって参

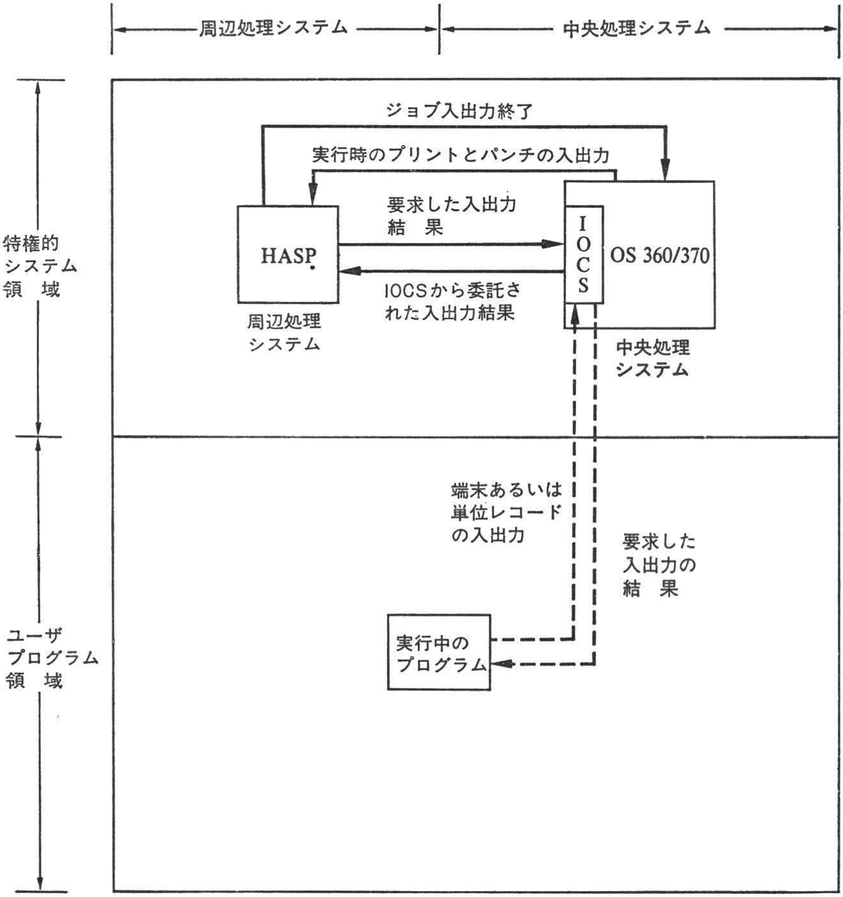


図 3.27 HASP および OS 360/370 に対する中央および周辺処理システムの役割の記憶配置

照できるユーザの料金計算の情報によって決まる．このようにして，ジョブの前処理と後処理は，HASP によって実行される．

仕事4.（すなわち，中央処理システムと交信すること）を行なうために，HASP は，中央処理システムの IOCS を端末や単位レコード機器に関連した入出力操作を扱えるものと置き換える．これらの型の入出力に対する実行中の(ユーザ)プログラムからの要求は，IOCS が受け付ける．しかし，この場合，IOCS がこの要求を処理するのではなく，HASP にこの要求を渡す．要求した入出力の結果が使えるようになったら，HASP はそれを IOCS に渡し，要求しているプログラムに戻る．HASP と OS 360/370 との基本的な関係を，図3.27に示す．

要約すると，ジョブ入出力に対する周辺処理システムについて，いくらか詳しく論じてきた．特殊化し工夫を凝らしたジョブ入出力システムは，オンライン処理システムの出現によって進められた．そこでは，ジョブを集め中間の記憶を利用するので，ジョブ入出力システムを**ジョブスプーリング・システム** (job spooling system)とも呼ぶ．

3.4　あ　と　が　き

一括処理システムの概括については，読者は，[17] と [18] とを見るとよい．オフライン一括処理システムは，一般的な概要は [14] に，詳細な概念については，[5], [8], [9], [10], [11], [13], [16], [24] に基づいている．オフライン一括処理システムを処理系および副処理系の階層と見なす考え方は，私のものである．しかし，これは，[4] で雄弁に語られている．オンライン一括処理システムは，特性化や抽象化の手助けにはならない．これは，大部分の材料を取ったオンライン一括処理システムに著者が深くかかわっていたことに，恐らく原因がある．このシステムは，[7], [15], [23], [25] の仕事であり，[3] によって提唱してきた．多重周辺システムの構成に対する HASP [19] の一般化については，[12] に述べてある．結合システムは，多重プログラミングの出現する前に，多大の興味をひき起こした．読者は，[2] と [20] に，興味ある提案を見つけることができる．プログラミング言語処理系の実現に関する問題は，[22] にある．

参　考　文　献

1. "An Introduction to the IBM 3790 Communication System," File No. 370-01, Form GA27-2767-0, First Edition, November 1973.

2. F. R. Baldwin, W. B. Gibson, and C. B. Poland. "A Multiprocessing Approach to a Large Computer System." *IBM Systems Journal* **1** (September 1962): 64.
3. J. W. Carr and N. S. Prywes. "Satellite Computers as Interpreters." *Electronics*, November 1965, pp. 87–89.
4. E. W. Dijkstra. "The Humble Programmer." *Comm. ACM* **15**, 10 (1972): 859–866.
5. R. T. Dorrance. "Design of an Integrated Programming and Operating System—Part IV: The System's COBOL Compiler." *IBM Systems Journal* **2** (September–December 1963): 322–327.
6. H. H. Goldstine. *The Computer from Pascal to von Neumann*. Princeton, N.J.: Princeton University Press, 1970.
7. D. K. Hsiao. "A File System for a Problem Solving Facility." The Moore School Report No. 68-33 (May 1968), The Moore School of Electrical Engineering, University of Pennsylvania. Also NTIS AD 671 826.
8. "IBM 7040/7044 Operating System (16/32K)—Programmer's Guide." File No. 7040-36, Form C28-6318-7, Eighth Edition, 1966.
9. "IBM 7040-7044 Operating System (16-32K)—Systems Programmer's Guide." File No. 7040-36, Form C28-6339-4, Major Revision, October 1965.
10. "IBM System/360 Basic Operating System—Programmer's Guide." File No. S360-20, Form C24-3372-6, Seventh Edition, September 1967.
11. "IBM System/360 Basic Operating System—System Control Program Logic Manual." Form Y24-5002-0, First Edition, December 1965.
12. "IBM System/360/370 Attached Support Processor System (ASP)." Version 2, System Programmer's Manual, Form GH20-0323-8, Ninth Edition, March 1971.
13. R. Larner. "Design of an Integrated Programming and Operating System—Part IV: The System's FORTRAN Compiler." *IBM Systems Journal* **2** (September–December 1963): 311–321.
14. G. H. Mealy. "Operating Systems." Rand Report No. P-2584, The Rand Corp., Santa Monica, California, May 1962, NTIS AD 603146. Excerpts appeared in *Programming Systems and Languages*, S. Rosen (Editor), New York: McGraw-Hill, 1967.
15. R. P. Morton. "On-Line Computing with a Hierarchy of Processors." Moore School Report No. 69-13, The Moore School of Electrical Engineering, University of Pennsylvania, December 1968.
16. A. S. Noble. "Design of an Integrated Programming and Operating System—Part I: System Considerations and the Monitor." *IBM Systems Journal* **2** (June 1963): 153–161.
17. Saul Rosen. *Programming Systems and Languages*. New York: McGraw-Hill, 1967, pp. 4–23.
18. Robert F. Rosin. "Supervisory and Monitor Systems." *Computer Surveys* **1**, 1 (March 1969).
19. T. H. Simpson, R. P. Crabtree, and R. Ray. *HASP-II/Houston Automatic Spooling Priority System With Remote Job Entry*. Program Order No. 360D-05.1.014, IBM, September 1968.
20. E. C. Smith, Jr. "A Directly Coupled Multiprocessing System." *IBM Systems Journal* **2**

(September–December 1963).

21. J. E. Thornton. "Parallel Operation in the Control Data 6600." *AFIPS Proc. FJCC* **2**, 26 (1964): 33–40.

22. Peter Wegner (Editor). *Introduction to System Programming.* London and New York: Academic Press, 1964.

23. R. L. Wexelblat and H. A. Freedman. "The MULTILANG On-line Programming System." *Proc. AFIPS 1967 SJCC* **30**: 559–569.

24. B. White and J. Trimble. "Design of an Integrated Programming and Operating System—Part VI: Implementation on the 7040/44 Data Processing System." *IBM Systems Journal* **3**, 1 (1964): 79–94.

25. M. S. Wolfberg. "Control, I/O, and Editing Functions in the Problem Solving Facility." The Moore School Report No. 65-31, The Moore School of Electrical Engineering, University of Pennsylvania (June 1965).

演　習　問　題

1. 3.2節で提案した階層的構造を考えて，システム構成での特定な処理の要求に基づいた注文式のオフライン一括処理システムを生成できるオフライン一括処理システム生成系を設計せよ．この生成系は，言語翻訳サブシステムのようなきちんと定義された処理機能の大規模な集合体になっていると仮定してよい．それにも拘わらず，この生成系は，どれだけ雑多な機能を一緒に持たなければならないかということを示さねばならない．

2. オンラインプログラミングの機能と，オンラインジョブ入出力機能との差異を，ソフトウェアの機能的要求とハードウェア要素と機器の支援という立場から詳細に論ぜよ．一方の機能をもち他方をもたないというシステムもあり得ることを示せ．例を述べよ．

3. オンラインプログラミングとオンラインジョブ入出力機能の両方をもつ現実のシステムを示せ．

4. オンラインシステムは，3つの部分に分けられるものと仮定する．第一の部分は，オンラインプログラミングもできる**フォアグラウンドジョブ**(foreground job)とよぶ高位優先度(入出力限界)ジョブを専門に扱う．第二の部分は，**第一バックグラウンドジョブ**(background-one job)とよぶ中位優先度(入出力限界)ジョブのためにあり，このジョブは遠隔投入端末を介して入力されオーバーレイを必要とする．最後の部分は，**第**

二バックグラウンドジョブ(background-two job) である低位優先度ジョブのためにあり，このジョブは，大抵計算限界である．

オンラインシステムにおけるジョブの実行とスケジューリングを，状態遷移図を使って示せ．ヒントとして，状態，遷移をひき起こす要因，および，図の概略を示しておく．

ジョブは，次の3つの状態をとると考えることができる．

実行中
停止
実行可能

a)　それぞれのジョブの型(すなわち，フォアグラウンド，第一バックグラウンド，第二バックグラウンド)に対して，前述の3つの状態を使って状態遷移図を書き，例を使いながら遷移をひき起こす要因を示せ．

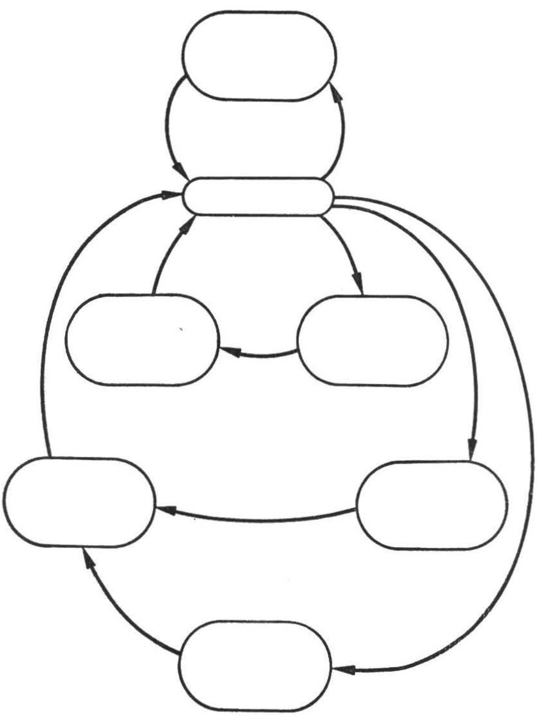


図 3.28

b)　**図3.28**に示した状態遷移図を下記に示した状態(s_i)と要因(f_j)とを使って完成せよ.

s_1　高位優先度(入出力限界)ジョブの実行可能リスト

s_2　中位優先度(入出力限界)ジョブの実行可能リスト

s_3　低位優先度(CPU 限界)ジョブの実行可能リスト

s_4　端末入出力のため停止しているジョブ

s_5　重ね合せ入出力と遠隔ジョブ入出力のため停止しているジョブ

s_6　ディスクあるいはテープ入出力のため停止しているジョブ

s_7　実行中

f_1　従属部分を要求(オーバーレイのため)

f_2　端末入出力を要求

f_3　ディスクあるいはテープ入出力の要求

f_4　オーバーレイの入出力が完了

f_5　端末入出力が完了

f_6　ディスクやテープの入出力が完了

f_7　(ユーザのジョブ制御カードに示されているかシステムが決めた)時間制限を超過

f_8　特定の時間周期に対して実行

5.　3.3.1.2節で提案した衛星計算機のオペレーティング・システムに対し，状態の遷移をひき起こす要因を書き出し，状態遷移図を作成せよ．また，3.3.1.3.1節に概略を示した中央計算機のオペレーティング・システムの割り込み処理系に対する状態遷移図を作成せよ．そして，これら2つの状態遷移図を結合して，衛星および中央システムの間のインターフェースを明らかにせよ.

6.　Rosen の教科書 [17] の第5部にあるオペレーティング・システムに関する歴史的論文をいくつか読み，それに対する感想を述べよ.

7.　Goldstine の計算機に関する本 [6] を読み，感想を述べよ.

8.　HASP におけるシリンダ探索アルゴリズムに対して行なった設計上の次の決定について説明せよ.

a)　近隣探索では，1番最近参照されたシリンダの(両側に対し)隣り8個だけのシリンダを探索する．何故8個か.

b)　すべてのモジュールの中で1番最近参照されたシリンダのすぐ隣

り（両側）のものをすべて探索し，次にその隣りというようにしないで，近隣探索アルゴリズムは，次のモジュールのシリンダをどれもチェックしないで，現在のモジュールの1番最近参照されたシリンダの(両側に対し)隣り8個をすべてチェックする．すべてのモジュールを一緒にしないで，一時に一つのモジュールで働くように探索を設計したのは何故か．

第 4 章

多重プログラミング

4.1 注　　意

多重プログラミングを学ぶためには，最初に，従来の実記憶を採用しているシステムを調べることから始めてみよう．それらのシステムの構成が一括処理システムの構成に似ているだけでなく，プログラムの構造や結合の規則なども一括処理のものと同じである．これらの相似性について，指摘するつもりである．

一方，もっと強力な多重プログラミング・システムを，タスクの概念に基づいて構成する．実行中のタスクの同期をとったり，共通資源に対してタスクを相互ロックするために，セマフォアと $\boldsymbol{p}$ および $\boldsymbol{v}$ 操作の研究が始まる．セマフォアには，タスク間で共用される資源が対応づけられる．その操作は，タスクの進行の転送，延期および発生をすることができる．

システムデッドロックという新しい現象が，資源が少なくタスクが頑固なときに起り得る．このために，システムデッドロックを定義するための条件を導こう．さらに，システムデッドロックを防止したり検出したりするアルゴリズムを提案し，デッドロックからのシステム回復について考える．

4.2 なぜ多重プログラミングか

前に述べたように，オフラインジョブ入出力と結合システムを使う目的は，一括処理システムにおける CPU の利用率を向上させ，ユーザに対してジョブの一巡時間(すなわち，ジョブの提出から戻るまでの経過時間)を減少させることであった．システムが遅く，小型または中型の計算限界のプログラムが大部分なので，CPU の計算の能力の利用率を高めジョブの一巡時間を高速化するために上記の技法を使えば，ある程度の成功を見ることができる．

最近のハードウェアの発達とユーザのプログラミングに対する要求の変化とを考えると，上記の技法は極めて不適当になってきた．CPU の内部処理の速度†は，大きく改善された．そして，ユーザは直接計算機システムにアクセス

† 注 CPU の内部処理速度は，CPU が機械命令を解釈し実行する速度で決まる．機械命令を解釈するためには，伝統的な計算機は，主としてノイマン型のプログラム内蔵方式なので，命令を主記憶から取ってこなければならない．正しい命令を持ってくるためには，命令の主記憶における番地を，(プログラムカウンタ PC の中に)保持し，(記憶番地レジスタ MAR を介して)記憶制御装置に与えなければならない．それが(記憶バッファレジスタ MBR を介して)もって来られたとき，主記憶の元の番地の中にある命令はそのままであり，命令の写しを一時的な作業レジスタの中に退避しなければならない．命令を主記憶から取り出し作業レジスタ内に置くために必要な時間を，記憶サイクルと名づける．命令の演算の型の識別と被演算子の検証は，次のようにして行なう．被演算子は主記憶の中の情報を参照していてもよいので，情報のとり出しを追加する必要性もあるかもしれない．したがって，CPU による命令の解釈は，1 記憶サイクル以上になり得る．この場合，複数個の作業レジスタを使う．

演算の型が正しく識別され，その被演算子が順次検証されれば，命令の実行フェーズが始まる．実行の間，関係する作業レジスタを使う．さらに，演算の結果は，主記憶に格納しなければならないかもしれないので，もう一つ記憶サイクルが必要になる．したがって，実行を遂行するために必要な時間は，演算のためのハードウェア回路の性能，レジスタのアクセス時間，および，追加の記憶サイクルにより決まる．

前の議論から，CPU の内部処理速度は，主記憶へのアクセスの頻度，主記憶のサイクル時間，演算に対するハードウェア回路の性能，レジスタのアクセス時間などにより影響を受ける．したがって，CPU の内部処理時間を改善するためには，主記憶アクセスの頻度を減らすこと，記憶サイクルを短かくすること，回路の性能を向上させること，レジスタのアクセスを高速化することをしなければならない．この改善を実現するような技術上の進歩がいくつかある．

1. 主記憶の技術．現在の技術では，磁気コアは，0.5 μsec のサイクル時間で，0.5 ¢/bit の価格である．少し短いサイクル時間を高い価格で実現できるけれども，磁気コアの大部分は，0.75 から 2.0 μsec のサイクル時間を，0.5 ¢/bit 以下の価格で実現している．

コア記憶は，価格が 0.25 ¢/bit 位で全部半導体かまたは半導体/バイポーラのハイブリッドの主記憶に徐々に置き換えられるであろう．

1 サイクルでもってくるビットの数は，**データ路**(data path)(並列に転送できるビットの数)の中と**記憶折り重ね**(memory interleaving)(異なった主記憶モジュールに干渉せずに同時にアクセスできる数)の多重度とから決まる．明らかに，データ路が広ければ，データバスをたくさん必要とし，折り重ねの多重度が高ければ，記憶をモジュール化して構成しなければならず，両方とも余分の費用がかかる．それにもかかわらず，記憶のサイクル時間を短縮し，記憶アクセスの頻度を減らす有効な方法は，これである．たとえば，データ路の幅が 64 ビットでサイクル時間が 0.75 μsec のとき，1 記憶サイクルで 64 ビットが持ってこれる．データ路が広ければ，解釈や実行のためにたくさんの情報を持ってこれる．さらに，記憶の折り重ねの多重度が 2 であれば，実効的な記憶サイクルは，0.40 μsec になり得る．

2. 高速記憶技術．高速レジスタの伝統的な使い方としては，UNIVAC 1108 のような番地付け可能なスクラッチパド記憶か，あるいは，IBM 370/165 で使われたような番地付け不可能なキャッシュ記憶がある．これらの記憶は，大部分バイポーラ型であり，記憶サイクルが 54～115 nsec，容量が 8 K～32 K バイト，価格が 3 ¢～6 ¢/bit という特性をもつ．もっと価格の高いものとしては，記憶サイクル 15 nsec のものが達成できる．番地付け不可能な高速記憶はプログラムには見えないので，プログラムは，高速に解釈し実行するために高速記憶に現在転送してきた情報の番地を追跡する必要はない．したがって，キャッシュ記憶を使うことが，番地付け可能なスクラッチパド記憶に徐々にとってかわりつつある．情報は，“固まり”になって，CPU が解釈し実行するために主記憶から高速記憶へ転送される．それゆえ，高速記憶の記憶サイクルが重要な役割を果たす．さらに，プログラムの参照が局所化されており(同一の固まりにまとまっており)，プログラムの命令が主記憶に格納する演算を少ししか含んでいなければ，得られる性能は，低速の μsec の記憶ではなく，高速の nsec の記憶によって左右されるであろう．

し，速い一巡時間を望むので，オンラインプログラミングが発展し，ジョブスケジューリングが考えられるようになった．

計算機システムに直接アクセスするためには，端末が必要になる．さらに，ディスクのような記憶機器が，ユーザによってオンラインに格納され検索される常駐型のデータやプログラムを格納するために必要になる．

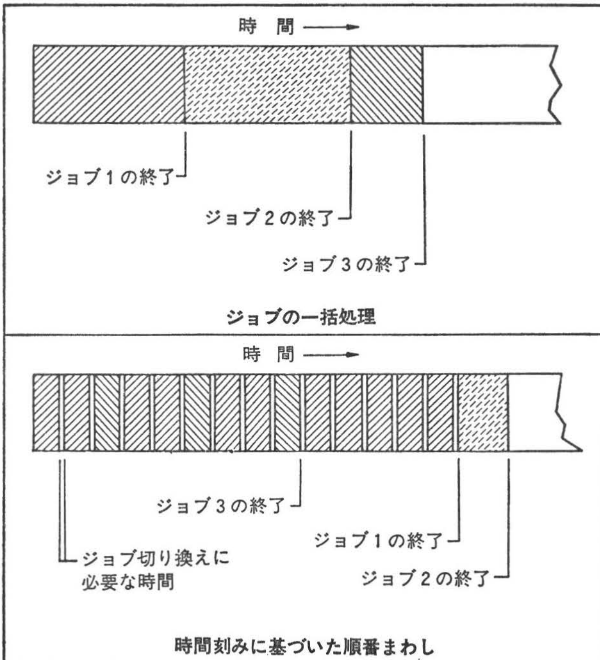
いろいろの大きさのオフラインジョブの一巡時間を速め，オンラインジョブのユーザに対して良い応答時間を得るためには，各ジョブは，処理時間の単位，すなわち，**時間刻み**(time slice)を与えられる．すなわち，ジョブの処理は，順番をまわす手続き(**図4.1**参照)に基づいて行なわれる．ジョブがその時間刻みを使い尽したか，入出力操作の完了を待たなければならないとき，そのジョブは，働きを止められ(すなわち，待ち状態にされ)，別の新しいジョブかすでに一部を処理されているジョブが，処理のために働かされる．このようにして，どの単一のジョブもシステムを支配することはできず，一つのジョブにおける入出力完了を待っているCPUの時間は，別のジョブの計算に使われる．さらに，複数個のジョブをシステムで実行可能に保っておけば，CPUの性能が良くなり，ジョブの一巡および応答時間が改善される．しかし，常にジョブの処理を可能にし，ジョブの一巡時間を減らすためには，ジョブの入力およびジョブからの出力は，さらに管理が必要となる．カード読取り機に対するジョブの入力のスタックと，穿孔機および印刷機に対するジョブの出力の記録を，たとえばディスクのような高速の入出力機器上に作成し，現在のジョブの実行がジョブの入出力の低速性によって極端に停滞してしまわないようにしな

(前ページより)

たとえば，80 nsecのサイクルで16 Kバイトのキャッシュ記憶をもち，1 μsecの記憶サイクルの主記憶を半メガバイト以上もつとき，実験的な研究によれば，性能は，80 nsecのサイクルの単一の記憶をもつシステムが達成するものの80%に相当するといわれている．したがって，キャッシュ記憶を使うと，大規模な主記憶が小規模なキャッシュ記憶のサイクル時間で働いているように見える効果を生み出す．

3.　回路技術．　回路技術は，離散型の機器から集積回路(IC)にかけて注目に値する程進歩してきた．エミッタ結合論理(ECL)やトランジスタートランジスタ論理(TTL)のような一般的なICの伝播遅れは，数年前20 nsecであったものが，nsecに満たないところまで改良されてきた．その間，計算機化した設計とテストにより，パッケージ化した論理回路(LSI)は，1チップにつき50から1000ゲートを有するようになった．たとえば，IBM 370/145の使用しているパッケージ化した回路のチップは，各チップごとに次に示すような特性をもっている．128個の記憶セルと，コード解説，ドライブとセンスの機能，出力信号の増幅などを含む46個の支援回路から成る174個の完全な回路である．

4.　マイクロコード技術．機械命令は，内部的には，マイクロコードの系列で作られている．機械命令の実行速度は，そのマイクロコードの演算の間に高度な並列性が得られれば，改善することができる．さらに，機械命令の機能的な能力は，マイクロコードが豊富であれば，増大することができる．



この図により，次のことがわかる．

1．時間刻みに基づいた順番まわしの処理では，小さいジョブは速い一巡時間であるのに対し，大きいジョブは比較的遅い一巡時間となる．
2．一括処理システムでは，後から提出したジョブは，事前に提出されたジョブすべてが終了するまで待たなければならない．
3．時間刻みに基づいた順番まわしの処理システムでは，後から提出したジョブであっても，処理の時間刻みをいくつか受け取れる．したがって，ジョブの中間結果を得，即時に応答することが可能である．
4．一つのジョブに対する現在の時間刻みが尽きて，別のジョブの処理が可能であるときには，各時間刻みの終りではジョブ切り換えのオーバーヘッドが存在する．

図 4.1 ジョブ処理

ければならない（第3章のジョブのスプーリングシステムに関する3.3.2.1.3節参照）．

さらに，働きを止めたジョブが占有していた主記憶領域を，働かし始めたジョブが使えるようにするために，ジョブを，後で動かすために二次記憶に転送しておかねばならないかもしれない．

これらの要求をすべて満たすと，過重な入出力動作になる．入出力機器の改

良がCPUの改良に追いついていなかったので，状況はもっと深刻である．たとえば，ディスクでは，転送率，アクセス時間，回転遅れにおける改善は，1.5から2の程度(次表に示してある)であったのに，CPUの内部処理速度は，脚注に述べたように，5〜10の程度であった．

ハードウェアの改良が，CPUの内部処理の速度と入出力機器の転送およびアクセス時間とに関して釣合いがとれず，また，ユーザからは速い一巡時間と直接アクセスが要求されるので，多重プログラミングを導入することが促進される．**多重プログラミング**(multiprogramming)とは，一つ以上のプログラムが主記憶に存在しかつ現在実行しているプログラムが完了する前に別のプログラムを動かしたり実行を再開するような技法を意味する．こうすれば，計算機システムの中で複数個のプログラムが並行的に実行でき，計算機の資源をもっと充分に使いこなせる．そのような資源として，たとえば，CPU主記憶，周辺装置などがある．システムで複数個のプログラムを実行すれば，次のような結果が得られる．

1. プログラムの入出力の要求を処理するとき，入出力操作の完了に対するCPUの待ち時間は，他のプログラムを実行することにより大幅に減少できる．それゆえ，CPUは，複数個のジョブがシステムによって多重プログラムされているとき，もっと有効に使える．さらに，たとえば，ジョブスプーリングシステムは，システムの“永久的な”ジョブとして走らせることができる．

2. ある時間刻みに基づいた順番まわし手続き(**図4.1**参照)を，次の目的で使える．(a)良い一巡時間を得ること．これは，一巡時間——もっと詳しく言えば，**応答時間**(すなわち，出力を要求してから出力が到着する

年	ディスク	転送率 (字またはバイト/秒)	アクセス時間 (msec) 最小/平均/最大	平均回転遅延 (msec)
50年代	IBM 1301 IBM 1302	90K 180K	50/120/180 50/120/180	17 17
60年代	IBM 2311 IBM 2314	156K 312K	25/75/135 25/60/135	12.5 12.5
70年代	IBM 3330	806K	10/30/50	8.3

までの経過時間)——が重要であるオンラインプログラミングの環境で特に有益である．(b) システム資源のもっと釣り合いの取れた使用をすること．すなわち，CPU と周辺機器を使用中の状態に保つこと．

4.3 ロード時再配置可能多重プログラミングシステム

多重プログラミングの定義は単純ではあるけれども，多重プログラミングシステムの実現には，多くの変形が見られる．多重プログラミングシステムの実現に関する考え方には，2つの流派があるように思える．一つは，プログラムの再配置をロード時にするような計算機システムで多重プログラミングシステムを実現すると良いと信じる一派，もう一つは，プログラムの再配置を実行時にする計算機システムで多重プログラミングシステムを実現すると良いと信じる一派である．前者は，より伝統的な計算機システムを使え，また，一括処理システムの設計において有益であったと証明された経験や方法を利用できるという利点をもつ．後者は，**記憶変換**(memory translation)という特別のハードウェア，特別な記憶の構成，管理およびプログラム結合を必要とする．

この章では，ロード時に再配置可能な多重プログラミングシステムについて集中的に扱おう．仮想記憶を使った実行時に再配置可能な多重プログラミングシステムについては，次章で扱うつもりである．

4.3.1 ジョブだけの多重プログラミング

最も単純な多重プログラミングのオペレーティング・システムは，一個以上のジョブは並行的に走れるが，一つのジョブの中のプログラムは，一時に一つづつ順次に処理しなければならないようなものである．すなわち，一つのジョブの中のプログラムは，多重プログラムはできない．一つのジョブのプログラムが，時間刻みの終了や入出力操作の完了待ちのために停められたとき，別のジョブのプログラムが，処理のために動かされる．その間，停められたプログラムは，主記憶内に残っているか，または，後で動かすために二次記憶機器に転送される．

これらのシステムでは，与えられたジョブ内でのプログラム間の多重プログラムをしなくてもよいので，設計や操作が比較的単純である．設計に際して考慮すべき多くの点は，一括処理システムで考えたことをいくらか修正したり拡張したりすればよい．例として，IBM 360 のテープ/ディスク・オペレーティング・システム(TOS/DOS)をあげる．

4.3.1.1　入出力制御ブロックと IOCS との構成

IBM 360 TOS/DOS における IOCS とその制御ブロックの構成は，入出力操作と IOCS に関する章で述べたことと非常に似ている．これらの相似点と相違点とを示してみよう．しかしながら，読者は，入出力から見て，これら2つのシステムの構成は似ているけれども，それらの術語はそうではないことに気を付けなければならない．

入出力機器への記号による参照を許すために，前に述べたように，機器に対して装置制御ブロックを作成した．装置制御ブロックには，記号装置に対して個々の入出力操作の進行を追跡するための情報がある．さらに，各入出力機器に対し，入出力機器の性質や現在の状態に関するすべての情報を置いておく機器制御ブロックが与えられている．IBM 360 TOS/DOS は，常駐型スーパーバイザに少ない記憶 (6～24 K) しか割り当てられないような構成の計算機に対して設計したものであるから，記憶を節約するように制御ブロックを単純化してあり，装置や機器の割り当てにいくらか柔軟性を欠いている．記号装置が使う機器の型やチャネルに関する情報（第2章で項目 U1 と述べているもの）を装置制御ブロックに置くことにより，オペレーティング・システムは，同一チャネルの似た形の使用可能な物理機器を動的にその装置に割り当てることが出来る．IBM 360 TOS/DOS では，装置-機器の割り当ては，オペレータか，ユーザのジョブ制御文で陽に指定して行ない，オペレーティング・システムが行なうのではない．オペレータは，記号装置の性質と使える機器を知っているので，お互に正しく割り当てることができるものと仮定する．したがって，この情報(項目 U1)は，装置制御ブロックには現在のところない．IBM 360 TOS/DOS で，装置に対し現在要求されている入出力操作に関する（第2章で項目 U2 とよんだ)情報は，**入出力要求待ち行列**(I/O Request Queue)または**チャネル待ち行列** (Channel Queue) とよぶ待ち行列へのポインタが対応する．多重プログラミングの環境では，同一の装置に複数個の要求が出されるので，待ち行列を使うことが要請される．チャネルコマンドや機器オーダの形式や実行に関する（項目 U3 および U4 とよんだ）情報は，入出力待ち行列の中の要求エントリのところに置いてある．同一の装置に対する異なった要求は異なった入出力操作になりチャネルコマンドや機器オーダも異なってくるので，この情報は論理的なものである．入出力の誤りに対する回復については，（項目 U5 の）誤り回復ルーチンの番地を保持していなければならない．IBM 360 TOS/DOS は標準的なテープ入出力回復ルーチンを提供し，ユーザが自分自身の回復ルーチンを使うという選択はさせないので，いろいろの回復ルーチンの番地

を保持しておく必要はない．それゆえ，**テープ誤りブロック**(Tape Error Block；略して **TEB**)とよぶ表のエントリへのポインタだけがあり，また装置制御ブロックの中には回復操作に関する統計をもっている．

直接アクセス機器の分割に関する情報(項目 U 6 と U 7)は，**ジョブ情報ブロック**(Job Information Block；略して **JIB**)とよぶ別の表に置いてある．装置制御ブロックとその装置に割り当てられた記号名とを関連づけるためには，記号装置名表へのポインタが必要である．このポインタ(項目 U 8)は，入出力要求待ち行列のエントリのところに置いてある．IBM 360 TOS/DOS では，記号装置名表を，**論理装置ブロック**(Logical Unit Block；略して **LUB**)とよぶ．

要約すると，装置に関する制御情報で JIB，TEB および入出力要求待ち行列に入っていなかったものは，**物理装置ブロック**(Physical Unit Block；略して **PUB**)とよぶ装置に対するブロックの中に集められている．第2章で並べた装置制御ブロックに入っている多くの情報は，PUB と TEB の中にある．もっと詳細にいうと，機器が使用中かどうか，機器のつながっているチャネルが使用中かどうか，および，機器が使用の要求を出されているかどうかを示す(第2章で D 1 といった)フラッグは，PUB にある．(項目 D 2, D 3, D 4, D 5 および D 8 は，それぞれ，U 2, U 3, U 4, U 5 および U 8 と同じ意味である．これらは，D 6 と一緒に，TEB の中に置かれていた)．機器のスケジューリングのためには，同一のチャネルにつながっている機器は，ポインタ(D 7)によりまとめてある．IBM 360 TOS/DOS では，PUB のエントリをチャネルの番号の順に並べることによって行なっている．PUB の各エントリは機器に対応するので，同一のチャネルにつながっている機器は，同一のチャネル番号によってまとめられる．

上の議論からわかるように，IBM 360 TOS/DOS では，機器制御情報に結合されていない装置制御情報だけを，JIB に入れておく．したがって，基本的には，JIB は，簡略化した装置制御ブロックをあらわし，PUB は，機器制御ブロックをあらわす．

論理装置ブロック(LUB)は，基本的には，第2章で定義した記号装置名表である．定義によれば，ブロックの各エントリにより，記号装置名と装置および機器制御ブロックとの対応づけができる．装置と機器制御ブロックは，ここでは JIB と PUB とで実現しているので，LUB のエントリの中にこれらのブロックに対するポインタがある．装置の記号名は，LUB の中のエントリの位置によって陰に定義される．たとえば，(図 **4.2** の)第 **1** のエントリは，SYSRES に，第2は，SYSRDR に対応する．

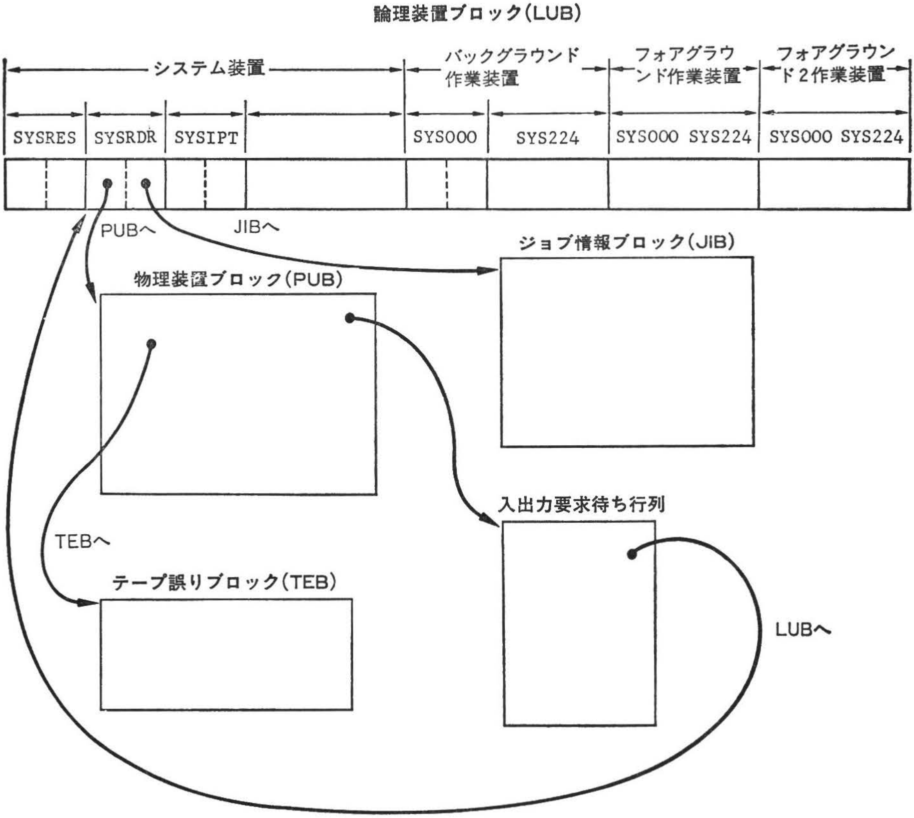


図 4.2　入出力制御ブロック

IBM 360 TOS/DOS における入出力制御システムの構成は，第2章で述べたものと似ている．しかし，リンキングローダで採用したバッファ割り当てアルゴリズムはない．したがって，余分の主記憶を(ユーザプログラムをロードした後で)追加のバッファとして入出力操作に割り当てる機能はない．

入出力の要求は待ち行列にするので，入出力操作実行系のチャネルスケジューラは，待ち行列の中の入出力操作を実行のためにスケジュールする．

待ち行列のエントリを結合して，与えられたチャネルがそれに対する入出力操作を次々と実行して，利用度を上げることができるようにする．ラウンドロビンの手続きでは，待ち行列の先頭のエントリは，機器が使用中でない限り，最初に実行される．チャネルが使用中であれば，そのチャネルにつながってい

るすべてのエントリを飛ばして，他のチャネルにつながっている最初のエントリをスケジュールする.

4.3.1.2　ファイルの保全性と入出力デットロックに関する考察

第2章では，ディスクのような直接アクセス機器を複数個の装置に分割し，それぞれに記号装置名をつけることを述べたと思う．記号装置名を参照することにより，ユーザの入出力要求に応じて機器のアクセス機構をその装置に割り当てられた区画に位置づけすることができる．そして，その区画に対する以後の読み込み/書き出しを，入出力操作実行系が行なえる．しかし，多重プログラミングの環境では，もとの位置に対する読み込み/書き出し操作が実行される前に，別のユーザジョブからの入出力要求によって，機器のアクセス機構が別の位置に動かされてしまうかもしれない．それゆえ，前のジョブに対する入出力要求の再開始をする必要がある．そうしないと，以後の読み込み/書き出し操作は，誤った区画に対して行なわれ，分割における**ファイルの保全性**(integrity of file)に反してしまう.

要求の処理を再開始するために，以後の読み込み/書き出し操作のためにアクセス機構の位置づけをする余分のアクセス時間が必要になる．さらに，新しい入出力の要求により，現在の読み込み/書き出し操作を実行する直前に，同じアクセス機構を異なった区画に位置づけしてしまうという状況も起るかもしれない．読み込み/書き出し操作を実行するためには，アクセス機構をもう一度再位置づけしなければならない．言いかえれば，これら2つの入出力要求は，お互に要求されている読み込み/書き出しの操作を実行することを妨げ合っている．読み込み/書き出し操作を何もしないで，永久にアクセスの競争をし合うという絶望的な状況も生ずるかもしれない．このような状況を**入出力デッドロック**(I/O deadlock)という．ファイルの保全性と入出力デッドロックの問題は，4.5節の資源共用とシステムデッドロックのところで，もう一度調べてみよう．ここでは，このような問題があるということに注意しよう．さらに，保全性とデッドロックの問題は，主として多重プログラミングの出現によって起ってきたものであることに注意しよう．複数個の入出力要求が同時にあらわれるのは，多重プログラミングの環境だからである．入出力機器やチャネルが共用されているので，アクセスの競争と共用資源の使用により，保全性が破壊され，資源の上でデッドロックが生ずる.

ユーザの情報の保全性を保護するために，入出力操作実行系は，**最初にユーザの入出力要求を受け取ったときに**，チャネルコマンドや機器オーダから成る

CYLINDER SEEK 操作を**必ず**実行する．CYLINDER SEEK 操作により，ユーザプログラムから要求している装置において正しいシリンダにアクセス機構を位置づけることができる．さらに，ユーザは，CYLINDER SEEK 操作を含む物理的入出力命令を出すことは許されない．すなわち，入出力操作実行系は，チャネルコマンドのプログラム(語)を調べて，CYLINDER SEEK 操作を引き起こすようなすべてのコマンドを認めない．

アクセス機構の位置づけを最小にし多重入出力要求によるデッドロックを避けるためには，アクセス機構をその位置に“ロック”することが一つの方法となる．それを実現するために，入出力操作実行系は，与えられた入出力要求に対するチャネルコマンドや機器オーダをつないで，シーク操作の後の読み込み/書き出し操作が終了するまではそれに割り込ますことができないようにする．こうすれば，アクセス機構は，現在の入出力要求が完全に終了するまで，再位置づけされることはあり得ない．

4.3.1.3　オペレーティング・システムの階層構造

図 3.3 を見ると，一括処理オペレーティング・システムの階層構成を一般化すれば，ジョブだけの多重プログラミングをする機能をもった多重プログラミング・システムの設計に適用できることに気が付く．一般に，ジョブの集団を管理しその集団の一つのジョブから別のものへの遷移を監視する複数個のスーパーバイザがある．したがって，一般的な構成では，多重プログラムできるジョブの数は，オペレーティング・システム内に実現されているジョブスーパーバイザの数に等しくなる．さらに，同一のスーパーバイザの監視の下にあるジョブでは，スーパーバイザの下にある構成が基本的には一括処理システムなので，多重プログラミングすることができない(**図 4.3** 参照)．

IBM 360 TOS/DOS の構成は，上で概説したものに似ている．貧弱な計算機構成のために，かなりの簡単化が行なわれている．システム生成時に，IBM 360 TOS/DOS は主記憶を 4 つの領域に分割できる．一括処理システムの場合と同じように，システム領域は，常駐型制御プログラムと核とから構成されている．他の 3 つの領域は，ユーザプログラム用である．それらは，**バックグラウンドプログラム領域**，**第一フォアグラウンドプログラム領域**，および，**第二フォアグラウンドプログラム領域**とよぶ．非常駐型制御プログラムは，制御の性質に応じて，システム領域かユーザプログラム領域のどれかに動的にロードされる．3 つのプログラム領域があるので，IBM 360 TOS/DOS は，3 つのジョブスーパーバイザを各領域一つづつ作る．各ジョブスーパーバイザは，そ

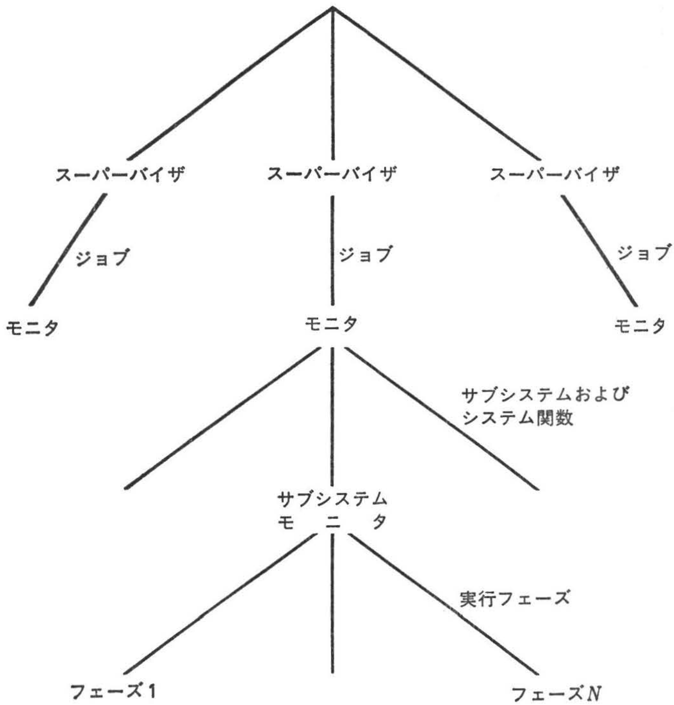


図 4.3　多重プログラミング・オペレーティング・システムの階層構造

のジョブを処理のために持って来れるので，オペレーティング・システムは，3つのジョブの多重プログラミングの機能をもつ．多重プログラミングに対するジョブのスケジューリングは，勿論，常駐型制御プログラムのジョブスケジューラによって実行される．ジョブの中のプログラムが入出力の操作の完了を待っていたり，割り当てられた時間を使い切ってしまったとき，ジョブスケジューラは，次に述べる優先度規準に基づいて他のジョブ内のプログラムを選び，そのプログラムに制御を渡す．ジョブの優先度は，ジョブが入っている領域によって決まる．バックグラウンドプログラム領域は，最低の優先度，第一フォアグラウンドは最高の優先度をもつ．優先度が固定されているので，第一フォアグラウンドプログラム領域のジョブがCPUの使用を支配してしまうことも起り得る．このため，オペレータは，第一フォアグラウンドおよび第二フォアグラウンドプログラム領域で実行するジョブを，注意深く選択しなければ

ならない．一般に，これらのジョブは，極端に入出力限界であり，実時間を考慮するという制限がある．

各プログラム領域のスーパーバイザは，基本的には，前に第3章で述べた一括処理スーパーバイザである．主な相違点は，次の通りである．(1) スーパーバイザを構成する制御プログラムは，大部分非常駐型で対応するプログラム領域へロードして実行する．(2)サブシステム間および実行フェーズ間の通信(第3章にも述べてある)のための領域も，プログラム領域にある．各スーパーバイザに対する通信領域は一つである．

さらに，バックグラウンドプログラム領域と，フォアグラウンドプログラム領域との間に相違点がある．後者は，プログラミング言語サブシステムを支援することはない．これらのプログラムをコンパイルしたりアセンブルするときには，ユーザは，そのプログラムをバックグラウンドジョブとして実行しなければならない．スーパーバイザを設計する際に，フォアグラウンドでのコンパイルやアセンブルの機能を止めてしまう理由は，次の通りである．

1. 機器の制限．コンパイラやアセンブラは，そのコンパイル時やアセン

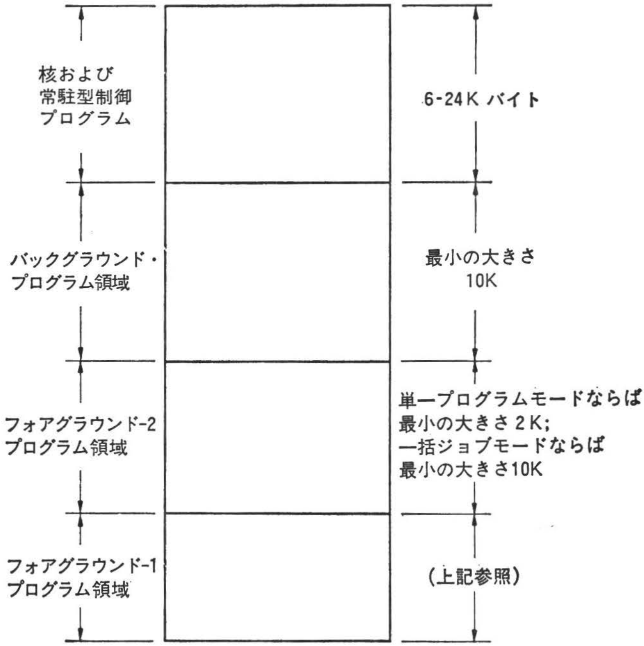


図 4.4　IBM 360 TOS/DOS の主記憶配置

ブル時に，中間ステップのために作業空間としてテープやディスク装置を使うので，一つ以上の区画でそれを同時に実現するほど充分な周辺機器がないかもしれない．(IBM 360 モデル 30 から 50 のシリーズでは，ディスクやテープ装置から考えた周辺装置の価格は，CPU の価格を上まわることもあるかもしれない)．

2. 主記憶の制限．常駐型制御プログラムに割り当てられている主記憶は，6K から 16K バイトである．このような量の記憶では，入出力制御システム (IOCS) に凝って，オペレータやユーザの陽な指定を介さずに IOCS が動的に入出力機器を割り当てるようにすることは不可能である．自動的な入出力機器割り当ての機能をもった同じような構成のもっと進んだ多重プログラミングのオペレーティング・システムは，少くとも 32K バイトの主記憶を必要とする．例として，IBM 360 の固定数のタスクの多重プログラミング (MFT) オペレーティングシステムがある(文献参照)．

4.3.1.4 要　　約

IBM 360 TOS/DOS の設計思考は，大部分高度な一括処理システムから引き出してきていることに注目しよう．それゆえ，この多重プログラミング機能をジョブだけに制限していることは驚くに当らない．多重プログラミングの環境で必要なジョブの選択と入出力要求のスケジューリングは，システムの中に実現してある．入出力機器の自動的割り当ては，多数の一括処理システムには見られるけれども，ここでは使わない．したがって，IOCS の大きさは，最小になっている．これは，機器に制限のある環境で走ることを予想しているので，フォアグラウンドのプログラムをコンパイルしたりアセンブルする機能を，オペレーティング・システムは提供していない．

4.3.2 ジョブおよびジョブ内のプログラムに対する多重プログラミング

ジョブだけに対する多重プログラミングと，ジョブだけでなくジョブ内のプログラムも含めた多重プログラミングは，少々違いがある．後者における進んだシステムの機能では，ジョブがシステムの資源を競争するだけでなく，一つのジョブ内のプログラムが同一の資源を互に競争し合うことを許している．

ジョブのかわりにプログラムを多重プログラミングの最小単位にすると，システムの資源に対してさらに大きく厳しい競争が生ずる．これは，システム資源が非常に制約を受けることのないようなシステム環境において特に役に立

つ.

ジョブはジョブ制御言語の文によって定義されるので，オペレーティング・システムは，多重プログラミングに対するジョブを容易に識別することができる．しかし，多重プログラミングに対するプログラムを識別するためには，オペレーティング・システムに対して追加の情報が必要となる．プログラムがサブルーチンを動的に呼び出すとき，そのプログラムは，オペレーティング・システムに対しサブルーチンが呼び出すプログラムと多重プログラミングをするかどうかを知らせなければならない．この情報は，プログラムの準備の段階でジョブ制御文によって与えられているかもしれないけれども，サブルーチンによってさらに呼び出されるかもしれないルーチンについて情報を与えることは難しい．

このため，ユーザプログラムがオペレーティング・システムと通信をして，これらのルーチンが呼び出しているプログラムと多重プログラミングのモードで動くのか一括モードで動くのかを教えるという手段を追加しておく．

これにより，ユーザのプログラムは，(1) サブルーチンを二次記憶からロードまたは実行のために動的に呼び出すこと，(2) 呼び出されるサブルーチンが呼び出しているサブルーチンと多重プログラミングをするかどうかをオペレーティング・システムに伝えること，(3) サブルーチンが呼び出すプログラムと多重プログラミングで走る場合，プログラムの残りの部分とサブルーチンとの同期をとること，ができる．

動的に呼び出すサブルーチンやデータを備えるために，オペレーティング・システムは，主記憶割り当ての手段――伝統的な(実)主記憶を利用しているシステムに対して永久的に課せられる必要条件である――を提供しなければならない．この機能があれば，ユーザプログラムは，そのサブルーチンやデータに対する追加の主記憶を要求することができる．主記憶が少なくなると，オペレーティング・システムは，次のことをしてユーザからの主記憶の要求を返す．

1. 空間が空いていないという指示を与え，ユーザのサブルーチンのローディングと実行とを中止する．
2. オペレーティング・システムは，プログラム優先度が低い別のプログラムを後で実行するために二次記憶へ転送(スワップアウト)しようと努力する．こうして空いた空間は，それを要求していたものへ与える．

2. の機能をもつシステムは，**スワッピング**(swapping)機能をもつものであ

る．オプションとしてスワッピング機能をもつ現存の多重プログラミングシステムとして，IBM 360 MVT システム——可変数のタスクをもつ多重プログラミング——がある．

4.3.2.1　タスクおよびサブタスクから見たオペレーティング・システムの構成

一括モードで実行しているプログラムと多重プログラミングのモードで実行しているプログラムとを区別するために，**タスク** (task) という概念を導入する．タスクとは，多重プログラミングされるものである．一方，一括処理の必要なプログラムは，常に順次的に処理される．多重プログラミングの機能をもったシステムでプログラムを順次的に処理するために，**ジョブステップ** (job step)という考え方を使う．ジョブのジョブステップは，順次的(すなわち一括的)に処理するものである．一般に，一つのジョブは，一個以上のジョブステップから構成され，各ジョブステップは，一個以上のタスクから構成される．ジョブステップ内のタスクはシステム資源を並行的に競争できるけれども，与えられたジョブのジョブステップは，システム資源を順次的方式で利用しなければならない．それにもかかわらず，異なったジョブのジョブステップは，同一のシステム資源を並行的に競争することができる．したがって，異なったジョブのタスク(同一ジョブの異なったジョブステップではないもの)は，同一のシステム資源を並行的に競争することができる．

プログラムを呼び出し，呼び出したプログラムと多重プログラミングをするためのマクロ命令を**アタッチマクロ** (attach macro) という．呼び出したプログラムを**(主)タスク**，呼ばれたプログラムを**サブタスク**(subtask)とよぶ．すなわち，サブタスクは，(主)タスクがアタッチマクロを発すると，(主)タスクに付けられる．

アタッチマクロが使えれば，サブタスクは非常に簡単に作成できる．実際，MVT オペレーティング・システム全体は，サブタスクから構成されていると見ることができる．構成の頂点には，組込みの**主システムタスク**(main system task)があり，それは，どのタスクのサブタスクでもない．論理的には，主システムタスクは，コンソールオペレータ，コンソールコマンド，それに，コマンドを処理でき，アタッチマクロを発することができ，オペレータがコマンドを介して指示した適当なサブタスクを作ることができるプログラムから構成されている．主システムタスクが作ったいろいろの形のサブタスクには，入力機器からユーザの入力ジョブ列を読み込む**読み込みサブタスク** (reading subta-

sk)，ジョブの処理を監視する**開始サブタスク**(initiating subtask)，および，出力機器にジョブ出力を書き出す**書き出しサブタスク**(writing subtask)がある．ユーザジョブは，少くとも一つのタスクから構成されているが，それは，勿論，開始サブタスクのサブタスクであり，それはまた主システムタスクのサブタスクでもある．主システムタスクに付随している読み込み，開始，および書き出しサブタスクの数は，オペレータの指示により決まるものである．すなわち，MVTオペレーティング・システムは，システム資源とジョブの負荷とに基づいて読み込み，開始，および書き出しサブタスクの必要な数を合理的に推定することは出来ない．さらに，異なったサブタスクには異なった優先度を割り当てることはできるけれども，これらのサブタスクは，非同期的な方式で多重プログラミングをしている．

複数個のタスクの注意深い同期が必要となるある種のシステム資源を利用することは，重要である．たとえば，MVT内に複数個の読み込みサブタスクがあるとき，これらのサブタスクを同期させ，入力ジョブを読んできてジョブ列を作り出す入力機器(たとえば，カード読み取り機)が停まるのを最小にして，それを最高の速度で動かすようにすることは重要である．一方，サブタスクを作るのは容易なので，全体のシステム資源利用率を高めるようにオペレータが読み込み，開始，書き出しのサブタスクの混合率を変更することができる．

MVTとTOS/DOSの構成は，非常に違っている．前者はタスク指向に，後者はジョブ指向にできている．さらに，MVTは，タスクの動的構成をもつように作られている．与えられた環境で作り出されるサブタスクの数は，オペレータやユーザの必要度に基づく．特に，使える記憶に応じて記憶の区画が作れる†．それゆえ，読み込み，開始や書き出しサブタスクのようなシステム関数の数は，変更できる．TOS/DOSでは，サブシステムやシステム関数の階層は固定していた．システムは，余分な資源が使えるようにならなければ，3個以上の区画を実現することは不可能である．

4.3.2.2　実行中のタスクの同期

タスクは多重プログラミングをする積りなので，それらは，本質的には実行中は相互に独立である．記憶，機器やデータのような資源がタスク間で共用されているような多くの状況では，タスクの主要部分は他のタスクと多重プログ

† MVTに対する記憶区画の最大の数は16である．この制限は，IBM 360ハードウェアの記憶保護機構，すなわち，16個の保護“キー”しかないので相互排除的な領域は16個しか保護できないことから生ずる．一方，MVTオペレーティングシステムの構成は，そのような設計上の制限はない．

ラミングの環境の下で実行されてはきても，そのタスクの一部は他のタスクの結果に依存するかもしれない．陽に同期をとらなければ，タスクは残りの仕事を正しくやり遂げるための正しい結果を受け取れないかもしれない．このために，独立したタスクの同期をとるために，2つのマクロ命令を提供してきた．

タスク内の**待ちマクロ**(wait macro)では，他のタスクからの結果(またはそこへ割り当てた資源)を一つ以上使えるようになるまではタスクを遅らせることができる．**送信マクロ** (post macro) では，一つのジョブから他のタスクへ信号を送って結果(または資源)が本当に使えることを伝えることができる．これらのマクロ命令を使えば，タスクは，

1. 相互に通信すること，
2. 進行を協調させること，

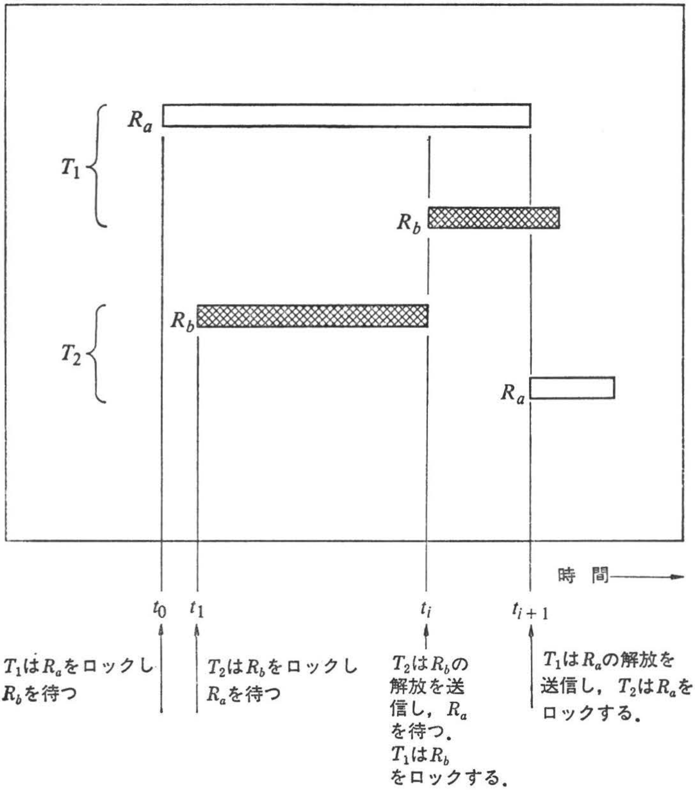


図 4.5 共通資源上でのタスク間の相互ロック

3. システム資源の上で，相互ロックすること，

ができる．システム資源とは，勿論，主記憶のような記憶媒体，ディスクのような物理機器，ファイルやライブラリサブルーチンのような論理的ものも含んでいる．

例として，**図4.5**に示すように，タスク T_1 と T_2 とがシステム資源 R_a と R_b との上で相互ロックしている状況を考えよう．T_1 と T_2 とは多重プログラミングをしているので，これらの資源の使用を，横棒によって並行的に表わすことができる．この例では，タスク T_2 は時刻 t_i で送信マクロを出して，資源 R_b の使用が完了したことを指示する．タスク T_1 はその資源を待ちマクロを介して待っていたので，資源 R_b は直ちに T_1 に割り当てられる．しかし，T_2 は，T_1 がまだ R_a の使用をやめていないので，R_a を待たなければならない．時刻 t_{i+1} までは，T_2 の待ちマクロにより，T_2 の作業の進行が遅れる．T_1 が t_{i+1} で R_a に関する送信マクロを出すや否や，T_2 の進行を再開する．この例では，これら2つのタスクは，共通資源の上で相互ロックをしていることに注意しよう．同期の機構がなければ，タスクは正しく進行することができないであろう．

送信および待ちマクロを使ってタスク間で相互ロックを陽に行なうことに加えて，タスクからオペレーティング・システムが制御している資源に対して出される要求も，相互ロックを生ずる．すなわち，オペレーティング・システムは，要求された資源がタスクに使えるようになる迄，そのタスクの進行を遅らせる．この場合，同期は，送信や待ちマクロを陽に使わないでオペレーティング・システムが行なう．

4.3.2.3 共通資源へのアクセスの制御

共通資源として，オペレーティング・システムの制御の下にはないデータを考えよう．一括処理の環境では，共通データに関するアクセスの制御は，全くユーザの責任に任されていた．すなわち，ユーザは，自分のプログラムの実行を(一括化するかオーバーレイするかして)整列して，各プログラムが共通データに事前に決めた順序でアクセスできるようにしなければならない．こうすれば，プログラムが共通データにアクセスすることは，先行する一括化あるいはオーバーレイ化したプログラムの実行がすべて完了したときに，初めて可能になる．ほとんどのオペレーティング・システムは，これをするために，ある種の誤り条件コードをジョブの一つのバッチのフェーズから同一ジョブ内の別のバッチのフェーズに渡す．これらの誤り条件コードは，オペレーティング・シ

ステムが管理しているジョブの共通データと考えてもよい.

多重プログラミングの環境では，オペレーティング・システムが共通データに対するタスクのアクセスを秩序立てて制御することが望ましい．たとえば，更新のためにデータを変更している間は，そのデータに対するアクセスをしてはならない．この場合，データはタスクによって専有的に使われており，他のどんなタスクもアクセスできない．一方，読み込みだけのデータに対しては，権利のあるタスクはいつでもアクセスすることができる.

このために，システムは，共通資源を定義したり名前をつけたりすることをユーザに許す機能を用意している．さらに，ユーザが名前をつけた資源へのアクセスを制御するために，**待ち行列挿入**(enq)および**待ち行列除去** (deq) の2つのマクロを提供する．待ち行列挿入マクロは，名前のついた資源の待ち行列に要求を入れる．待ち行列除去マクロは，資源の使用が終ったので，待ち行列から要求を除く．名前のついた資源は，独占的あるいは並列的使用のどちらでも定義できる.

4.3.2.4 資源割り当て戦略

ユーザジョブに資源を割り当てることは，オペレーティング・システムの機能である．もっと厳密に言うと，システムの開始サブタスクがジョブを監視しシステム資源に対するジョブの要求を処理するので，このサブタスクがシステム資源を割り当てる.

システム資源を到着してくるジョブに割り当てるときに戦略が必要になるのは，次の目的のためである.

- (S_1) システム資源の利用を最大にすること.
- (S_2) システム資源に対するアクセスの競争を最小にすること.
- (S_3) システム資源の保全性を守ること.
- (S_4) 開始サブタスクとそのジョブとの間のデッドロックを防止すること.

S_4 の項目が，非常に大切である．デッドロックは，あるジョブに割り当てられていたシステム資源を他のジョブに使えるようにすることを妨げるだけでなく，システムを完全に役に立たなくしてしまう．たとえば，図 **4.6(a)**では，**3**つの開始サブタスク I_1, I_2, I_3 と**3**つのシステム資源 R_a, R_b, R_c とがある．それぞれの開始サブタスクによって，R_a がジョブ**1**に，R_b がジョブ**2**に，R_c がジョブ**3**に割り当てられていたと仮定する．時刻 t_i には，ジョブ**1**はシステ

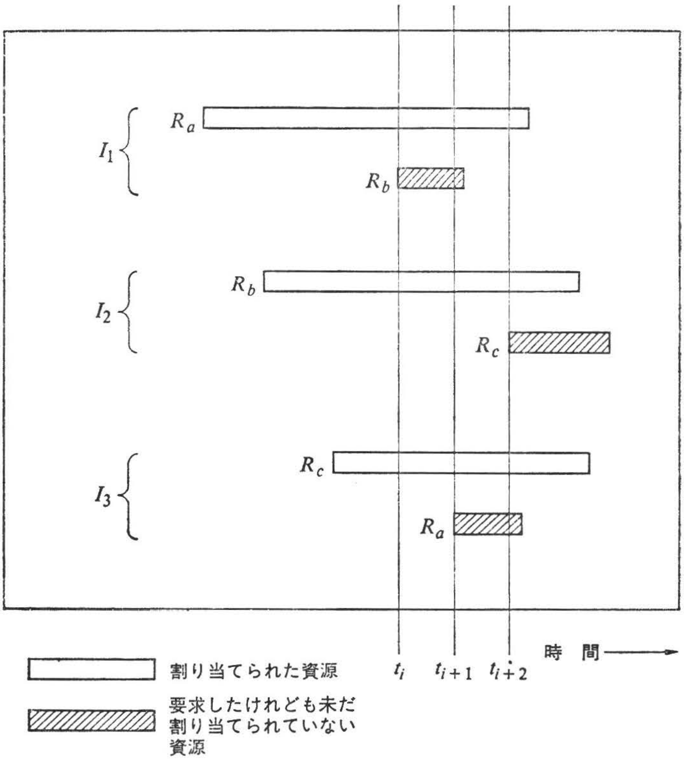


図 4.6(a)　資源上でのデッドロック

ム資源 R_b を要求する．R_b はジョブ2から解放されていないので，ジョブ1は待たなければならない．同様にして，ジョブ3は t_{i+1} に停止し，ジョブ2は t_{i+2} に停止する．時刻 t_{i+2} までには，明きらかにデッドロックが生じた．——3つのジョブが，すべてシステム資源に関するデッドロックで停止してしまった．したがって，システム全体は，役に立つ仕事を何もしていない．

開始サブタスクはシステム資源の割り当てに責任があり，また，システム資源の性質を良く知っているので，開始サブタスクがシステムデッドロックを防ぐ戦略を作ることが望ましい．

その戦略を作るときにデッドロックの防止は先ず第一に重要であるけれども，S_1, S_2, および S_3 に関する考慮も強調しなければならない．たとえば，ジョブは資源に対する要求をすべて最初に与えなければならなず，また，ジョブは，要求した資源がすべて使えるようになるまで開始されないという戦略を

採用すれば，システムデッドロックは起らないであろう．しかしながら，そのような戦略では，ジョブに割り当てられたシステム資源のうちのいくつかは，長い期間にわたり使用されずにおり，同じ資源を要求した他のジョブは長い間待たねばならないので，費用がかかる．あきらかに，この戦略は，S_1 を満足していない．一方，(1) もともと持っている資源のために追加の資源をとることが不可能なときには，追加の資源をもらう前に事前に得ていた資源をジョブは解放しなければならない，(2) ジョブは，まとめて資源が必要なときには，もとの資源を追加の資源と一緒に要求しなければならない，という戦略を採用するならば，システム資源の利用を大幅に向上することができるであろう．しかし，この戦略は，システム資源が主としてデータファイルの場合に健全には働かない．事前に得たデータファイルを解放し，新しいデータファイルと一緒にそのデータファイルを要求しても，データファイルが同じデータファイルを割り当てられた他のジョブによって汚される(すなわち，ファイルの保全性を犯す)ことは防げない．したがって，この場合，S_3 に反することになる．MVT システムにおけるシステム資源の割り当ては，個々のタスクのレベルでは起らない．そのかわりに，それは，ジョブステップのレベルで行なわれる．

開始サブタスクの中で実現している戦略は，次の通りである．

1. ジョブが要求しているシステム資源の型の決定は，システムが行なう．
2. 各型のシステム資源の割り当ては，その資源の性質を特に考慮して行なう．
3. 複数個の型のシステム資源が要求されたときには，事前に決めた順番の割り当てステップに従って行なう．

MVT システムでは，システム資源に次の 3 つの型がある．すなわち，ファイル(データセット)，機器および主記憶領域である．

ジョブに対するデータセットの割り当ては，ジョブの全期間に対してそれを割り当てればデータセットの保全性が最善に維持できるという信念に基づいて行なわれる．したがって，ジョブのタスクは，すべての必要なデータセットを獲得する迄は進まない——すなわち S_3 を満足する方法である．

ジョブに対する機器の割り当てでは，テープ駆動装置のような機器は，前の情報の集合が必要でなくなったとき，一つの情報の集合に対して使うという考えに基づいている．データセットとは異なり，機器は，保全性の問題はない．要求した機器すべてを，ジョブの全期間通してジョブに割り当てておく必要は

ない．さらに，ジョブの全期間に対してすべての要求した機器を割り当てると，ジョブの間でアクセスへの競争が生ずる．このため，機器の割り当ては，ジョブステップのレベルで，開始サブタスクが行なう．ステップの最後では，そのステップに割り当てられていたすべての機器を解放する．――すなわち，S_2 を満足しそうな方法である．

主記憶領域の割り当てに対する考え方は，機器に対するものと似ている．ジ

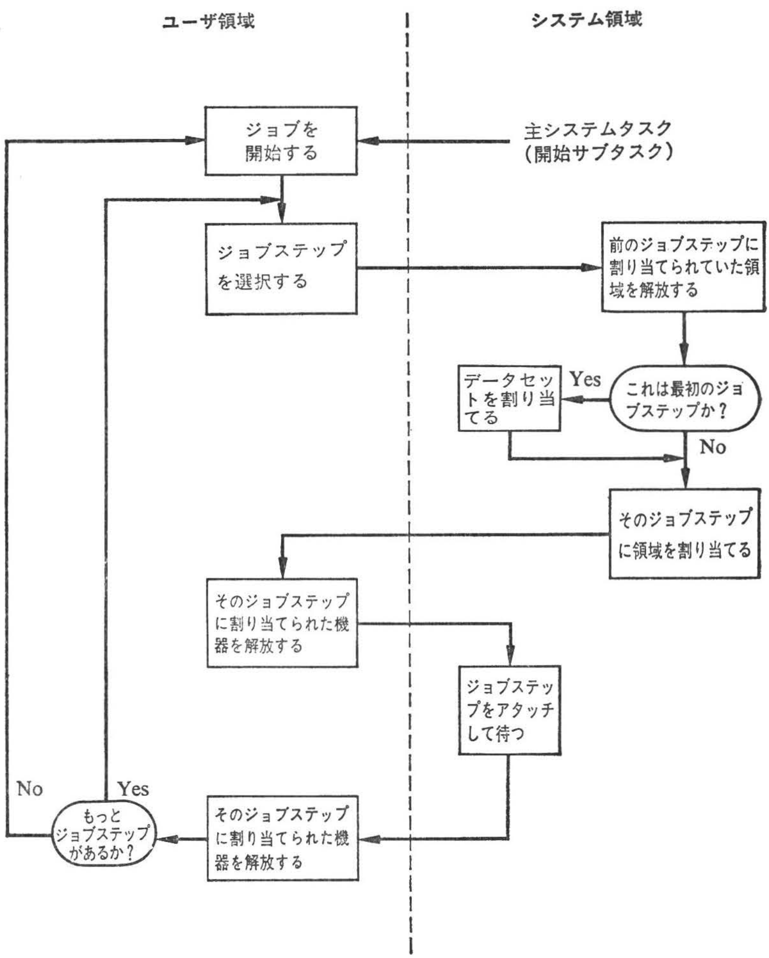


図 4.6(b)　資源割り当ての機構

ョブのレベルでは(領域とよぶ)記憶の区画を割り当てないので，システムは，ジョブのどれかのステップが要求する最大の規模の領域をそのジョブのために予約する必要はない．

すなわち，ジョブステップのレベルの主記憶領域の割り当てにより，開始サブタスクは，そのジョブステップまたは(開始サブタスクの大きさの方がジョブステップよりも大きい場合は)開始サブタスク自身に対して充分の大きさの主記憶を確保することができる．機器の場合と同じように，ステップの終りにはそのステップに割り当てられていた領域を解放し，次のステップに対する新しい領域を割り当てる．したがって，条件 S_1 および S_2 が満たされる．

ジョブによって3つの型の資源がすべて要求されたとき，次の戦略を使う．

1. 最初に，ジョブに対するデータセットを割り当てる．
2. 次にジョブステップに対する主記憶領域を割り当てる．
3. 最後に，ジョブステップに対する機器を割り当てる．

上に述べた割り当ての順序は，重要なことである．適切なデータセットがなければ，進行する必要はない．したがって，ジョブステップへの割り当ては，起らない．領域の割り当ては，システムの機器割り当てのルーチンが新しいジョブステップの領域で実行されるので，機器の割り当ての前に来る．領域をジョブステップに割り当てておけば，そのジョブステップに対する機器割り当ての要求を，そのルーチンによって処理することができる．

システム資源割り当て機構の図式を，**図4.6(b)**に示す．

4.3.2.5 制限された競争

前に注意したように，MVT システムは，システム機能を実行しサブシステムモニタとして働くために，同等のサブタスクを多数作ることができる．これらのサブタスクは，勿論，二次記憶にあるプログラムである．主記憶を保持し，異なった入力に対し複数個の同等なサブタスクがプログラムの一つの写しを使えるようにするために，これらのプログラムの大部分は，**再入可能型** (reentrant)にコード化してある．再入可能プログラムについては，4.6節で述べる．ここでは，それについて考えないけれども，次のことに注意しておくことは大切である．(1) 論理的には，複数個のサブタスクを参照しているけれども，これらのサブタスクの機能を実行しているプログラムは，物理的に主記憶には少ししかない．(2) オペレーティング・システムは，一つの再入可能プログラムが複数個のサブタスクによって共用されていることを知ることのでき

る機構が MVT の中で使える．この情報により，システムは，主記憶に既にロードされている再入可能プログラムと，そうでないものとの追跡をすることができる．再入可能プログラムと読み込みだけのデータセットとの相似点は，どちらも権利の認められたタスクによっていつでもアクセスされるということである．

最初に述べたように，多重プログラミングの設計の一つの目標は，システム資源の利用率を最大にすることである．この目標を達成するために，タスクという概念を導入した．もっと精密なジョブの単位の定義をすれば，もっと有効な資源の利用が可能になろう．少数の大規模なジョブがシステム資源を時々しか争わないのではなく，同一の資源に対して適切な要求と敏速な使用をするジョブ単位(タスク)がたくさんある方がよい．しかし，資源の共有を強調するような設計目標もある．さらに，実際には，システム資源は，滅多に余らない．事実，制限のある共用資源に多数の要求を出し，占有的に使えば，システムデッドロックが生ずる．デッドロックが生ずると，システム資源は何も使えなくなってしまう．このような状況では，システムデッドロックを回避するために，タスクを遅らせ，資源を再割り当てしなければならない．

4.3.3 結　　論

前節における多重プログラミングの研究の結果，システムは，一方で共通のシステム資源上でのタスク間の相互ロックをするため，また一方で，システムを無用のものにしてしまうシステムデッドロックを防止するために，効果的な同期の機構を提供しなければならない．

送信，待ち，待ち行列挿入，および，待ち行列除去のマクロは，これらのシステムのいくつかのものに実現されてきたソフトウェアの同期の機構である．しかし，多くの同期の機構が，最近ではハードウェアで実現されている．さらに，もっと原始的な同期機構を考案して，ハードウェアの実現が容易で前述のマクロをソフトウェアで作り出すのも容易にできるようにするという傾向が認められる．この目的のために，基本的で，“第四世代”の計算機で実現されている同期基本命令の組を論じよう．その使い方の例も示そう．

システムデッドロックを回避するために，それを生ずるための必要条件を識別しなければならない．これらの条件により，次に，システムデッドロックを防止し検出するための戦略とアルゴリズムの研究ができる．しかし，十分条件を導くのは難しい．十分条件を知らなければ，システムデッドロックの予測は難しい．

4.4 同期と相互ロックの機構

4.4.1 セマフォと同期基本命令

タスク間の同期は，(1) セマフォ，(2) 各タスクにおける **v** 操作および **p** 操作とよぶ命令の対によって達成できる．**セマフォ**(semaphore)は，整数の値を取るハードウェアレジスタか特定の記憶の場所である．同一のセマフォを使って同期をとるタスクが複数個あるので，システムは，そのセマフォに対してすべてのタスクの待ち行列を作る．

v 操作は，一つの被演算子をもつ機械命令である．被演算子は，セマフォの名前である．**v** 操作が実行されると，次のことを行なう．

1. セマフォの値が非負であれば，その値を一つ増加する．命令は完了し，この命令の実行が始まったときに動いていたタスクの次の命令へ制御が戻る．
2. セマフォの値が負であれば，(走っているタスクでなく)待ち行列の先頭のタスクが実行可能状態になる．セマフォの値を，一つ増加する．

p 操作も，被演算子としてセマフォの名前をとる機械命令である．**p** 操作が実行されると，次のことを行なう．

1. セマフォの値が0より大きければ，その値を一つ減らす．命令は完了し，この命令の実行が始まったときに動いていたタスクの次の命令へ制御が戻る．
2. セマフォの値が非正であれば，この命令の実行が始まったときに動いていたタスクは，待ち行列の中に入れられる．その値を，一つ減らす．

共通資源の上でタスクを相互ロックするためには，セマフォの値と **v** 操作および **p** 操作の実行を，次のように解釈する．

資源の供給者は，資源の使用可能なことを送信するために，**v** 操作を使う．資源の消費者は，資源が未だ使用可能でなければ，資源を待つために **p** 操作を使う．セマフォの値が0より大きくなったとき，資源が使えることを示している．各 **v** 操作は，豊富な供給を示すことができる．したがって，セマフォの値は，**v** 操作を実行するごとに，増加してゆく．使える資源がある限り，タスクの遅れが起らずに **p** 操作を実行できる．資源が消費されたときには，セマフォの値が減少する．セマフォの値が0になるときは使える資源がないことを意味するので，タスクの中で **p** 操作を実行すると，そのタスクが使える資源

はないので，タスクは停止してしまう．セマフォの値が負であれば，資源が直ぐに必要であることを示している．この場合，v 操作を実行しても，それは値を一つしか増やさないので，資源の増加がその欠乏を打消すことはない．このために，新しいタスクを動かさなければならない．これらが供給者であれば，資源が再び供給されて，セマフォの値は正になる．そうでなければ，資源の欠乏が厳しいか，あるいは，供給よりも消費が多いことになる．そのような状況では，p 操作を実行しているタスクは，待ち状態になる．

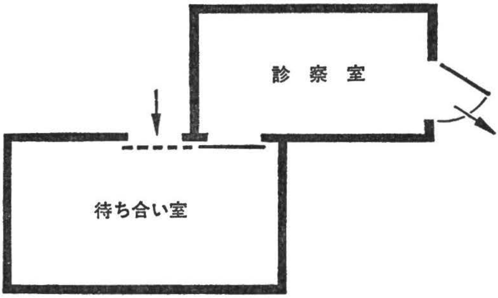
4.4.2　眠っている医者：一つの例

眠っている医者の話は，下に示した医院でのことである．医院は，待ち合い室と診察室とからできている．待ち合い室には，入口がある．入口の隣りには，医者の診察室への入口がある．両方の入口は，同じ引き戸を共用しており，一方が閉れば，もう一方が開いている．さらに，入口は非常に小さいので，一時には一人の患者しか通過できない．

医者が患者を診察してしまうと，待ち合い室の戸を開けて，もし患者が居れば，次の患者を呼び入れる．患者が一人もいなければ，医者は待ち合い室で眠ってしまう．

患者は，次のように振舞う．待ち合い室に患者がいれば，順番を待つだけである．しかし，眠っている医者を見つければ，彼を起こす．

この例では，2つのタスクがある．患者タスクと医者タスクである．セマフォは，待ち合い室の患者の数をあらわす．値が0より大きければ，医者タスクは，割り込みなしで進行できる．患者に診察室に入るように言うたびに，医者タスクは p 操作を実行することになる．その値が0であれば，p 操作の実行により医者タスクは動かなくなる．すなわち，眠ってしまう．



一方，患者タスクは，医者タスクがこの場面になるよりもずっと前から，活動している．この場合，セマフォは大きな正整数を簡単に蓄積できる．そうでなければ，医者タスクは，セマフォの値が1になる度に，動かされる．すなわち，医者は，新しい患者が空の待ち合い室に入ってくる度に起される．

4.4.3 要　約

多くの同期および相互ロックの機構が文献に提案されているけれども，前節に述べたものが，実際に計算機にハードウェアとして作られている．理論的には，ここで述べたものよりももっと原始的な機構を提案することもできる．たとえば，セマフォで，整数を2進値0と1とに制限することができる．p 操作および v 操作は，タスクの待ち行列を使わないように簡単化できる．これらの原始的機構は，もっと凝ったものと同じ効果をもたらすことができる．しかしながら，実用的理由と使用の簡単さとから，ハードウェアおよびソフトウェアの機構は，より工夫したものへ向かう傾向にある．

ここで学んでおくことは，これらの同期と相互ロックの機構は，次のような特徴をもつことである．

1. タスクが相互ロックした資源と進行中の事象とにセマフォの名前を対応づけられるようなセマフォ．
2. タスクにある資源が取られてしまったとかある事象が生起したことを知らせ，また，資源や事象がないときはタスクを遅らせることのできる p 操作．
3. タスクに資源が再供給されているとか事象が本当に生起したことを知らせ，また，資源や事象が少ないとき資源の供給者や事象の生産者の眠りを覚ますことのできる v 操作．

4.5 システムデッドロック

4.5.1 システムデッドロックの問題

システムデッドロックの問題を，IBM 360 TOS/DOS および MVT システムの場合について学んできた．この問題は，資源を共用しそれが乏しいときにだけ生ずるものである．この問題に対する解では，資源共用に不必要な制限をつけず，また，資源供給に異常な要求をしないでも，デッドロックを回避しなければならない．この問題と解とを説明するために，次の例を示そう．

4.5.1.1　銀行の問題

銀行の資金は，限られている．その資金を最大に利用するために，銀行は，お客にお金を貸したいが，このとき，個々のお客への貸金の上限は銀行の資金よりも低いけれども，貸金の上限の合計はこの資金を越えるかもしれない．

資金が10万ドルのとき，次のような状況が起これば，必ずデッドロックが存在する．

お　客	貸金上限 (1000ドル)	現在の貸高	信用残高
c_1	80	41	39
c_2	60	21	39

さて，この状況では，銀行には，38000ドル［100－(41＋21)＝38］の現金が残っている．もしも，どちらかのお客が1ドルも返さないで信用残高満額を要求しようとすると，その要求を満たすだけのお金は銀行にはない．すなわち，銀行は，もう一方の貸金を止めるか，新しい要求を拒否するかしなければならない．どちらにしても，銀行の評判は，悪くなる．

4.5.1.2　銀行の問題の解

評判を守るために，銀行は，デッドロックが回避できるように貸金を認める条件として次のことを採用する．

1. お客は，銀行に対し，事業の運営の中で借金の上限を最初に伝えなければならない．
2. 事業の運営のためにお客が得たすべての借金に対し，事業は，ある(固定した)時間内に完了し，完了のあかつきには借金を返してくれるという保障がある．
3. お客から要求された最初の借金は，その要求した借金がそのお客に対する上限を越えていなければ考慮の対象とする．残りは信用残高であり，以後の事業の運営において追加の借金として，同じお客から要求することができる．
4. お客が自分の上限および信用残高以内の借金を要求したときは，その借金が認められることを期待すべきである．しかし，銀行が即座にお金を貸してくれなくても，怒ってはならない．それにも拘わらず，銀行は，借金の要求を処理してくれ，しばらくすれば，お客はお金を貸して

貰えるであろう.

5. 銀行は，他のお客が事業運営が終り次第すぐに借りたお金に利息をつけて返してくれるので，お客に対して認めた貸金を実現できるということに確信を持っている.

6. 銀行は，期待する借金が銀行の資金を越えないようなお客なら誰でも受け入れてくれる.

お客の要求する貸金を進める前に，銀行は次のような手続きをとる.

銀行は，一定の資金 CAPITAL をもつ．新しいそれぞれの客 i は，借金の上限 CEILING(i)を銀行に伝える．それぞれの客(すなわちすべての i)に対して，次の関係が成り立たなければならない.

$$\text{CEILING}(i) \leq \text{CAPITAL}.$$

お客 i(すなわち，すべての i)が借金 LOAN(i, j)を申し込んだときは，いつも銀行は，申込み時に次の関係が成立しなければならないことを確かめなければならない.

$$0 \leq \sum_{j} \text{LOAN}(i,\ j) \leq \text{CEILING}(i).$$

言い換えれば，未決済の（$j-1$ 個ある）貸金すべての和とお客の現在の借金の要求との和は，そのお客の借金の上限より低くなければならない.

したがって，お客 i に対する借金の信用高 CREDIT(i)は，

$$\text{CREDIT}(i) = \text{CEILING}(i) - \sum_{j} \text{LOAN}(i,\ j)$$

から得られる.

さらに，銀行は，どれだけの現金 CASH が手元にあるのか知っている.

$$\text{CASH} = \text{CAPITAL} - \sum_{i} \sum_{j} \text{LOAN}(i,\ j)$$

であるから，あきらかに，

$$0 \leq \text{CASH} \leq \text{CAPITAL}$$

が成り立たなければならない.

承認した貸金を実施するかどうかを決めるために，銀行は，貸金を実施したならば生じうる状況を主として調べる．この状況が“安全”であれば，銀行は貸付を進める．そうでなければ，銀行はそれをしない.

状況が安全かどうかを決定するのは，すべてのお客の事業活動が終了すると

保証できるかどうかを調べることにつながる．その手続きは，少くとも一人のお客が現金を越えない借金の信用高をもっているかどうかを決定することから始まる．もしそうであれば，このお客は，自分のすべての活動を完了することができる．次にその手続きでは，第一のお客がすべての活動を完了しすべての借金を返したと仮定して，残りのお客を調べる．状況が安全であるとは，すべての事業活動が完了できることを意味する．すなわち，銀行は，その現金をすべて返して貰える．このような状況では，貸付を進めるための現金が足りないために，活動が無限に遅らされるということはない．

銀行の資金に対してシステム資源を，お客に対してジョブを，お客の信用の上限に対してジョブが要求する資源の合計を，また借金に対してジョブに割り当てた資源を代入すれば，銀行の問題は，多重プログラミングのデッドロック問題を直接相似をとったものになる．これによれば，各ジョブステップの開始のとき，すなわち，お客が新しい借金を申し出たときに，資源割り当てがどの位"安全"か調べられなければならないことがわかる．

4.5.2　システムデッドロックの特徴づけとその防止

システムデッドロックは，次の条件を使って特徴づけすることができる．

1. **相互排除条件**——タスクは，それが要求する資源を排他的に使いたいと要望する．
2. **待ち条件**——タスクは，追加の資源を待っている間，既に割り当てられた資源は保持している．
3. **非割り込み条件**——資源は，その使用が完了するまでそれを持っているジョブから強制的に取り上げることはできない．
4. **環状待ち条件**——タスクの環状鎖(circular chain)があって，その中の各タスクは鎖の次のタスクから要求されている資源を一つ以上持っている．

実際，これらは，システムデッドロックの状態を定義するための必要条件である．4.3.2.4節では，IBM MVTシステムにおける資源割り当て戦術がデッドロックの状況を回避することができたと述べたことを思い出そう†．この戦術におけるデータセットの割り当て法，すなわち，すべての必要なデータセ

† 勿論，計算機システムの外側でデッドロックの状態になり，いくつかのジョブが開始できないということは考えられる．

ットを得るまではタスクは進行できないというのは，明きらかに待ち条件を除去している．主記憶領域および機器の割り当てに対するこの戦術——すなわち，タスクは元来持っている領域や機器を解放してからもう一度追加の資源と一緒に要求しなければならない——は，非割り込み条件を消去している．最後に，資源の割り当てを順次的順番で行なう（すなわち，最初はジョブレベルのデータセット，2番目にジョブステップレベルでの主記憶領域，そして3番目にジョブステップレベルで機器の割り当てを行なう）戦術は，実効的に環状待ち条件を除いている．

しかし，相互排除条件を拒否することは難しい．たとえば，4.3.2.3節では，待ち行列挿入(enq)および待ち行列除去（deq）マクロを名前のついた資源を独占的に使うために用いると，相互排除条件が成立してしまう．

しかしながら，システムデッドロックの防止は，前述の4条件のうち少くとも一つを拒否すれば達成できる．

4.5.3 システムデッドロックの検出と回復

$r_1, r_2, \cdots, r_n$ を資源の型，w_1，w_2，$\cdots$，w_n を各型の資源の数を表わすものとする．任意の時刻 t において，タスク T_i に割り当てられている型 r_j の資源の数を p_{ij} で表わす．また，既にタスク T_i に割り当てられているもの以上にタスク T_i から要求されている型 r_j の資源の数を，q_{ij} で表わす．**割り当て**(allocation)および**要求**(request)行列を $P=((p_{ij}))$ と $Q=((q_{ij}))$ で表わし，T_i に割り当てた資源および T_i から要求される資源を示す行ベクトルをそれぞれ P_i および Q_i であらわす．さらに，割り当て行列 P の列を加えると，型 r_j の資源の割り当てた総数が得られる．すなわち，$\sum_{i=1}^{m} p_{ij}$ は，m 個のタスクに対して割り当てた型 r_j の資源の総数になる．$V=(v_1, v_2, \cdots, v_n)$は**使用可能資源**ベクトルで，その j 番目の要素($v_j \leq w_j$)は，現在使用できる型 r_j の資源の数を示す．ここで，

$$v_j = w_j - \sum_{i=1}^{m} p_{ij}$$

であることに注意しよう．

すなわち，型 r_j の資源で割り当てたものと使用可能なものとの和は，システム内にあるその型の総和に等しくなければならない．以下では，$\vec{0}$ は，要素がすべて0であるような(行)ベクトルをあらわす．また，x, y がベクトルのとき，$x \leq y$ が成り立つ必要十分条件は，x と y の対応する要素の対でこの関係

が成り立つことである．

下に示したアルゴリズムは，まだ未了のタスクのすべての可能な順序づけを単に考えるだけで，デッドロックを見つけるように設計してある．時刻 t において，システムの状態は行列 P および Q から得られると仮定し，また，V も与えられていると仮定する．さらに，時刻 t においてタスク i に割り当てたおよび要求された資源の行を，それぞれ $P_i(t)$ および $Q_i(t)$ で表わし，また，時刻 t での使用可能資源ベクトルを $V(t)$ で表わす．

1. $W \leftarrow V(t)$ という初期設定をする．すなわち，時刻 t における各型の資源の数は，V の中に示してある使用可能なもので決まる．P 内のすべての行は，最初から“マークなし”と仮定する．
2. $Q_i(t) \leq W$ となるようなマークなしの行 i を P 内で見つける．すなわち，要求した資源があるようなタスクをアルゴリズムは見つけようとしている．もしそのようなタスクが見つかれば，ステップ3.へ行く．そうでなければ，アルゴリズムはここで終了する．
3. $W \leftarrow W + P_i(t)$ とし，i 番目の行をマークし，ステップ2.へ戻る．

P 内のすべての行がマークされれば，タスクの系列 $\{T_i\}$ が存在して，この系列に基づいてタスクを(行がマークされた順序に)実行すれば，資源に対するデッドロックは生じない．一方，アルゴリズムの終了時にマークされていない行があるならば，デッドロックが存在する．もっと詳しく言うと，マークのない行は，まさしくデッドロックしたタスクである．したがって，このアルゴリズムは，デッドロックの検出に使える．

デッドロックが検出可能なので，次にデッドロックから回復する方法を考えることができる．デッドロックから回復する一つの方法は，(その行にはマークがついていない)デッドロックしたタスクをすべて中止することである．

このアルゴリズムは，銀行の問題に対する解の中で使った手続きに非常に似ている．唯一の違いは，銀行の問題では，一つの型の資源(すなわち，お金)しかなかったのに対し，計算機システムは n 個の型があること，また，銀行の問題では，銀行はお客を失うという危険は冒したくないのに対し，計算機システムでは，タスクはしばしば中止させても良い．タスクの中止を避けるためには，各タスクに対し，使える資源を事前に分配してしまうこと，タスクの処理が，資源が制限されているためにかなり大幅に遅れることを考慮することに同意すれば良い．

4.5.4 結　　論

デッドロックという現象を，例を使って説明した．デッドロックの状況を作り出す必要条件を述べている．資源の要求を大きさの順に並べて，各タスクに要求している型の資源の勘定を付随させることにより行なうデッドロック検出のアルゴリズムの概略を述べた．このアルゴリズムでは，タスクを中止することにより，デッドロック回復の解も得ることができる．

4.6 多重プログラミング環境におけるプログラムの構成

多重プログラミングの環境においては，共通のサブルーチンを複数個のプログラムが呼び出すかもしれない．主記憶を倹約しサブルーチンの追加の写しを読み込んだりロードしたりする際の入出力とCPUの動作を最小にするために，主記憶内の同一の写しを，すべての呼び出すプログラムが共用しなければならない．サブルーチンの共用に際しては，次のような問題が生ずる．

サブルーチンは，前の呼び出しに対する実行が完了する前に，別のプログラムから呼び出されるかもしれない． 終了する前にサブルーチンに進入されると，次の事項に関する情報が失われたり変更されてしまったりする可能性がある．

a) 呼び出しプログラムへの戻り番地，
b) パラメータの並び，
c) 作業領域，
d) プログラムの論理，
e) 機械のレジスタの内容のような操作環境．

上記の問題を解決する応急的な方法は，一方で，サブルーチンのプログラムの約束として他の呼び出しの進入を許す前に完了のチェックをすることにし，他方で，呼び出すプログラムでは連続した呼び出しを調べることにする．こうすると，現在呼び出しているプログラムに対するサブルーチン全体の実行が完了しなければ，サブルーチンに入ることは出来ず他の呼び出しプログラムが連続して呼び出すことが出来なくなる．そこで，呼び出しプログラムをサブルーチンの完了まで待たせておくべきか，あるいは，サブルーチンの写しの追加を主記憶に持ってくるべきかという疑問が生ずる．前者の場合，サブルーチンを待つことによりプログラムの性能が低下する．後者の場合，処理時間と主記憶の損失が生ずる．したがって，呼び出しプログラムによるサブルーチンへの進入が並行的にでき， かつ，上記の a)から e)に並べた変更される情報を退避す

ることができるような解が必要になる．

多重プログラミングの環境の中で共用するためには，共通サブルーチンは次のような方法で構成する必要がある．

1. サブルーチンを一つの手続き部と一つ以上のデータ部とに分けて記述する．
2. 手続き部は，サブルーチンの命令と，多分いくつかの定数とからできている．しかし，この部分の命令も定数も，実行中に変更されてはならない．
3. データ部は，呼び出しプログラムへの戻り番地，パラメータの並び，作業領域，および，汎用レジスタの内容からできている．さらに，サブルーチンの呼び出しごとにデータ部がある．
4. 呼び出しプログラムとサブルーチンとの間には，データ部から記憶を得，またそこへ情報を退避できるという約束をしておく．

上記のサブルーチンの構成により，多重プログラミングの環境において複数個のプログラムから並行的あるいは順次的に呼ばれても，サブルーチンは呼び出しプログラムごとに同一の機能を実行することができる．そのようなプログラムの構成は，**再入可能型**(reentrant)プログラム構成といい，プログラムの手続き部を**純**(pure)手続きという．

大規模なプログラムに対しては，プログラムの純手続きによりかなりの記憶を節約できる．たとえば，FORTRAN コンパイラが再入可能的にプログラムされていれば，すべてのユーザの FORTRAN プログラムは，同一のコンパイラを共用できる．すなわち，ユーザの FORTRAN プログラムは，コンパイラの(入力)データ部になり，コンパイルされたプログラムは，同じコンパイラの(出力)データ部になる．コンパイラは純手続きで構成しているので，現在のプログラムのコンパイルは他のプログラムによって偶然影響を受けることはない．しかしながら，記憶の節約は，主記憶の節約を意味しているわけではない．主記憶が小さければ，FORTRAN コンパイラのような大規模なプログラムに対しては，同時にはコンパイラの一つの実行フェーズしか主記憶には持ってこない．異なったプログラムは，コンパイルの異なった段階（したがってフェーズ）にいるかもしれないので，同一の実行フェーズがすべてのユーザの FORTRAN プログラムで必要になることは殆んどないであろう．したがって，実行フェーかズら見てコンパイラの写しをまだ多数使っている．IBM 360/370 MVT のような大規模記憶をもったシステムに対しては，共用のために再

入可能型プログラムの全体を主記憶の中に持っていることができる．この場合，システムは主記憶内の再入可能型プログラムを知ることができ，それにより異なったユーザのプログラムが再入可能型プログラムの手続きをサブルーチンとして呼ぶことができるような手段を与えなければならない．IBM 360/370 MVT では，そのような手段を LINK PACK とよぶ．どちらの場合にも，純手続きを使うとかなりの二次記憶が節約できる．プログラムの論理はその実行中変更することはできないので，プログラムの実行が途中で終っていても，同一のプログラムの写しを多数二次記憶に退避させる必要はない．たとえば，時間刻みを使い切ってしまった結果実行を一時的に停止させるために，プログラム全体と入力および出力データを二次記憶へ転送する(スワッピングと呼ぶ)ことが必要であるが，今や付随する入力と出力データのスワッピングをするだけなので，スワッピング時間を減少させるだけでなく二次記憶も節約できる．

4.6.1 再入可能型プログラムの結合の約束

結合(linking)とは，呼び出しプログラム P_i と呼ばれたサブルーチン P_j との間の制御の交換を意味する．結合は，呼び出しプログラムの中で**呼び出し系列**(calling sequence)という命令の集合を実行することによって始まる．そして，この結合は，呼ばれたサブルーチンの中でそれぞれ**入口初期設定**(entry initialization)および**出口戻り**(exit return)という 2 つの命令の系列を実行することにより維持され完了する．この約束では，呼び出しプログラム P_i は，次のことを与えなければならない．

1. P_i が他のプログラムから呼ばれる場合には，レジスタ A に入れたデータ部 D_i の番地 a_i.
2. レジスタ B に入れたパラメータ並びの番地，レジスタ C に入れた戻り番地，およびサブルーチン P_j への飛び越しを含んだ呼び出し系列．

サブルーチン P_j に入ってくるときには，次のことを行なうために入口初期設定の系列を出さなければならない．

3. そのデータ部 D_j にすべてのレジスタの内容を退避すること．
4. レジスタ A 内に D_j の番地 a_j を与えること．

サブルーチン P_j を出るときには，次のことを行なうために出口戻り系列を実行しなければならない．

5. D_j に退避してあるすべてのレジスタの内容を回復すること．

6. 特に，レジスタ A 内に番地 a_i を回復すること．

7. レジスタ C を介して，呼び出しプログラム P_i に飛び越すこと．

この約束では，レジスタ A は現在制御しているプログラムのデータ部の番

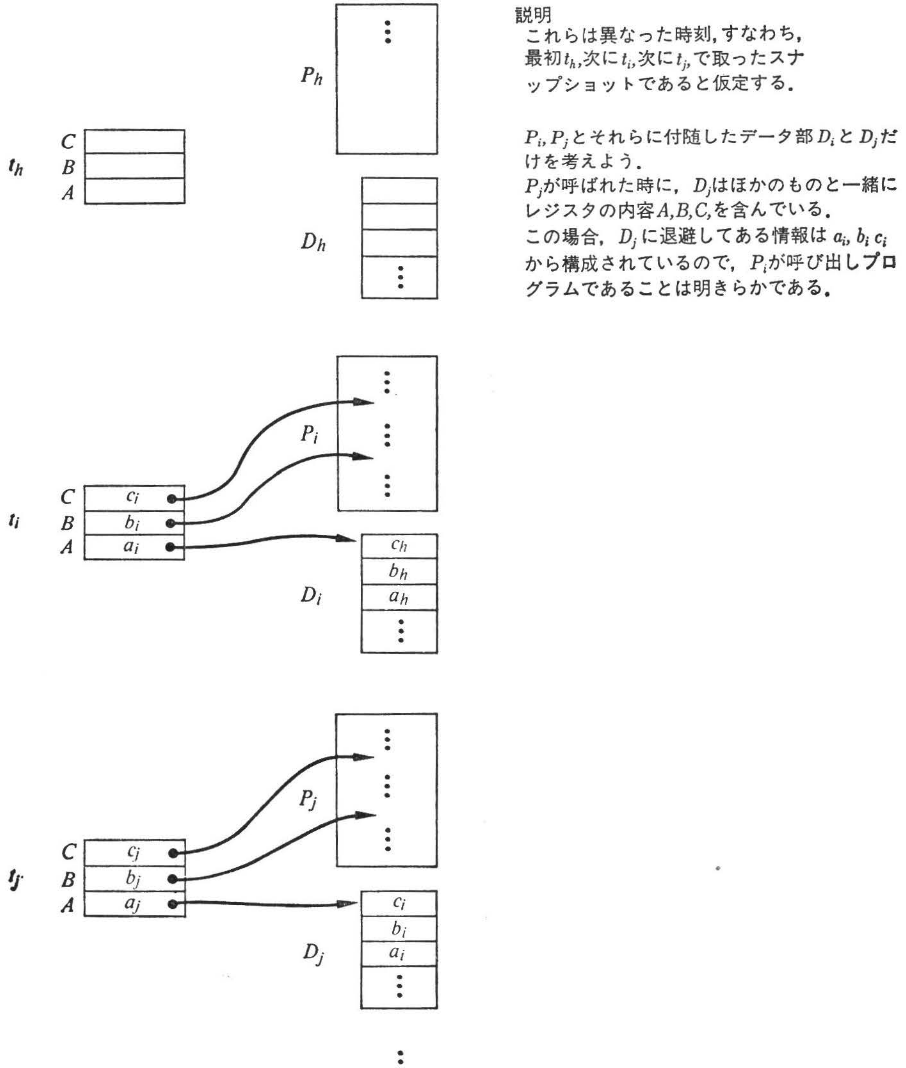


図 4.7　結合の約束

地を常にもち，レジスタ B は呼び出しプログラムのパラメータ並びの番地を保持し，レジスタ C は戻り番地に使われる．さらに，呼ばれたサブルーチン各々は，それが入られたときにすべてのレジスタの内容をそのデータ部に退避し，それが戻る直前に同一の内容を回復する．データ部に対して領域を動的に割り当てるために，呼ばれたサブルーチン P_j は REQUEST-MEMORY のマクロ命令を出すことができる．その命令により，割り当てられた領域の先頭番地 a_j が返される．P_i と P_j の関係およびレジスタ A, B, C の内容について，図 4.7 に示す．

4.6.2 再入可能型サブルーチンの結合の約束の例

IBM 360 および 370 のアセンブリ言語を使って，呼び出しプログラム P_i の基本的な呼び出し系列と，呼び出されたサブルーチン P_j の入口初期設定と出口戻り系列とを，下記に示す．さらに，呼び出されたサブルーチンのデータ部 D_j も，説明のために示しておく(図 4.8)．サブルーチンの入口と呼び出しプログラムのパラメータ並びの番地を宣言する擬似命令は，ここにはない．

呼び出し系列(P_i)

L	15, ADCON	レジスタ 15 にサブルーチン入口番地を入れる．
LA	1, PLIST	レジスタ 1 にパラメータ並び番地を入れる．
BALR	14, 15	レジスタ 14 に戻り番地がある．

入口初期設定(P_j)

STM	14, 12, 12(13)	レジスタ 14, 15, 0–12 の内容を D_j に退避する．
LA	12, DJ	レジスタ 12 にデータ領域の番地 a_j を入れる．
ST	12, 8(13)	データ領域の a_i に番地 a_j を置く．
ST	13, 4(12)	データ領域の a_j に番地 a_i を置く．
LR	13, 12	レジスタ 13 に a_j を入れる．

出口戻り(P_j)

LM	14, 12, 12(13)	レジスタ 0–12, 14, 15 を回復する．
L	13, 4(0, 13)	データ部の番地を，呼び出しプログラムから与えられたものと置き換える．この命令は，呼び出しプログラムが別のルーチンか

語	偏差	内容
1	0	使用せず
2	4	呼び出しプログラムのデータ部の番地 a_i (後方リンク).
3	8	そのデータ部の番地 a_j (P_jが P_kを呼ぶならば，呼び出されたプログラム P_kが格納する.)（前方リンク).
4	12	戻り番地(レジスタ14の内容)
5	16	エントリポイントの番地(レジスタ15の内容)
6	20	レジスタ0
7	24	レジスタ1
8	28	レジスタ2
9	32	レジスタ3
10	36	レジスタ4
11	40	レジスタ5
12	44	レジスタ6
13	48	レジスタ7
14	52	レジスタ8
15	56	レジスタ9
16	60	レジスタ10
17	64	レジスタ11
18	68	レジスタ12
(D_jの残りの部分)		

図 4.8　データ部 D_j

ら呼ばれたかもしれないので必要になる．

BR　　14　　レジスタ 14 にある番地により戻る．

4.7 あ と が き

IBM 360 TOS/DOS システムにおけるファイルの保全性と入出力デッドロックの問題は，[2] で論じている．IBM MFT と MVT とに関する情報はたくさんある——[12]，[13]，[14]，[15]，[16]，[17]——けれども，システムデッドロックの問題に関する研究は，描写的である．ここに述べた情報は，主として [10] から得た．同期と相互ロックの機構は [7] と [16] で提案され導入されたが，同期と相互ロック機構の最初の理論的取扱いは，[8] に見られる．

そこでは，セマフォの p および v 操作という術語が提案されている．このために，計算機製造業でも，この術語を使うようになっている．しかし，4.4.1 節で提案した機構は，[8] で提案したものよりも幾らか工夫を凝らしてある．眠っている医者および賢明な銀行家の話は，[8] から始まっている．

システムデッドロックの最初の認識は，[8] と [10] とに見られる．都合が良いので，システムデッドロックの特徴づけ，防止，検出や回復については，[3] に基づいている．

システムデッドロックに対する十分条件の研究は，ここでは触れていない．読者は，[9] と [11] を参照するとよい．

参考文献

1. B. W. Arden, B. A. Galler, T. C. O'Brien, and F. H. Westervelt. "Program and Addressing Structure in a Time-Sharing Environment." *J. of ACM* **13,** 1 (January 1966): 1–16.
2. G. Bender, D. N. Freeman, and J. D. Smith. "Function and Design of DOS/360 and TOS/360." *IBM Systems Journal* **6,** 1 (1967).
3. E. G. Coffman, M. J. Elphick, and A. Shoshani. "System Deadlocks." *Computing Surveys* **3,** 2 (June 1971): 67–78.
4. C. J. Conti, D. H. Gibson, and S. H. Pitkowsky. "Structural Aspects of the System/360 Model 85—I General Organization." *IBM Systems Journal* **7,** 1 (1968): 2–14.
5. P. A. Crisman (Editor). *The Compatible Time-Sharing System—A Programmer's Guide,* Second Edition. Cambridge, Mass.: The MIT Press, 1965.
6. A. J. Critchlow. "Generalized Multiprocessing and Multiprogramming Systems—Status Report." *Proc. of FJCC,* 1963, pp. 107–126.
7. J. B. Dennis and E. C. Van Horn. "Programming Semantics for Multiprogrammed Computations." *Comm. of ACM* **9,** 3 (March 1966): 143–155.
8. E. W. Dijkstra. "Co-operating Sequential Processors." *Programming Languages—NATO Advanced Study Institute,* Edited by F. Genuys. New York: Academic Press, 1968, pp. 43–112.
9. A. N. Habermann. "Prevention of System Deadlocks." *Comm. of ACM* **12,** 7 (July 1969): 373–377, and 385.
10. J. W. Havender. "Avoiding Deadlock in Multitasking Systems." *IBM Systems Journal* **7,** 2 (1968): 74–84.
11. R. C. Holt. "Some Deadlock Properties of Computer Systems." *Computing Surveys* **4,** 3 (September 1972): 179–196.
12. "IBM System/360 Operating System—Introduction to Control Program Logic," Program Logic Manual, Form &28-6605.
13. "IBM System/360 Operating System—Control Program with Option 2," Program Logic Manual, Form Y27-7128.
14. "IBM System/360 Operating System—Fixed Task Supervisor," Program Logic Manual, Form &28-6612.

15. "IBM System/360 Operating System—MVT Control Program Logic Summary," File No. S360-36, Form Y28-6658-1, Second Edition, January 1968.
16. "IBM System/360 Operating System—MVT Supervisor," File No. S360-36, Form No. Y28-6659-3, Fourth Edition, November 1968.
17. "IBM System/360 Operating System—Supervisor and Data Management Macro-Instructions," File No. S360-36, Form No. C28-6647-0, First Edition, February 1967.
18. "IBM System/370 Principles of Operation," GA22-7000-3, Fourth Edition, January 1973.
19. W. C. McGee. "On Dynamic Program Relocation." *IBM Systems Journal* 4, 3 (1965): 184–199.
20. T. H. Simpson, R. P. Crabtree, and R. O. Ray. "HASP-II/Houston Automatic Spooling Priority System with Remote Job Entry." IBM Program Information Department, Hawthorne, N.Y., Order No. 360D-05.1.04, September 1968.

演 習 問 題

1. (E. W. Dijkstra) 次に示す条件と例題手続きを考えよう．どの手続きがどの条件を持つかを見つけ，その理由を説明せよ．

説　明

(a) CPU の制御を実行中に他の手続きに切り替えてはならないとき，手続き部分は**危険**(critical)であるという．

(b) mbegin と mend を使うと，mbegin と mend の対の間にあるタスクは，他の対のものと多重プログラムができる．

(c) 手続きは ALGOL で書き，計算機システムは一台の CPU しかもっていない．

条　件

(a) 一つのタスクが危険部分の外側で走っているとき，他のタスクを停めてしまう可能性をもつことはない．そうでなければ，この条件は不必要に制限が強すぎる．

(b) 危険部分の現在の実行が完了する前に，その危険部分の実行を生ずるような条件が起る．

(c) 実行中のタスクの間で，相互に停止し合うような状況を作り出す条件が存在する．

```
"begin integer c1,c2;
        c1: = 1; c2: = 1;
        mbegin
        task 1: begin A1 : c1: = 0;
                      L1 : if c2 = 0 then goto L1;
                           critical section 1;
                           c1: = 1;
                           remainder of cycle 1;
                           goto A1
                end;
        task 2: begin A2 : c2: = 0;
                      L2 : if c1 = 0 then goto L2;
                           critical section 2;
                           c2: = 1;
                           remainder of cycle 2;
                           goto A2
                end
        mend
end".

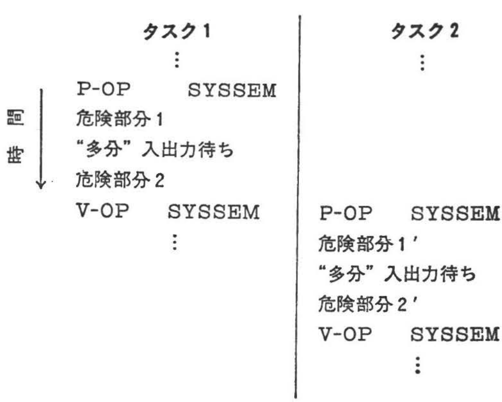
"begin integer c1,c2;
        c1: = 1; c2: = 1;
        mbegin
        task 1: begin L1 : if c2 = 0 then goto L1;
                           c1: = 0;
                           critical section 1;
                           c1: = 1;
                           remainder of cycle 1;
                           goto L1
                end;
        task 2: begin L2 : if c1 = 0 then goto L2;
                           c2: = 0;
                           critical section 2;
                           c2: = 1;
                           remainder of cycle 2;
                           goto L2
                end
        mend
end".

"begin integer turn; turn: = 1;
        mbegin
```

```
            task 1: begin L1 : if turn = 2 then
                         goto  L1;
                               critical section 1;
                               turn: = 2;
                               remainder of cycle 1;
                               goto L1
                    end;
            task 2: begin L2 : if turn = 1 then
                         goto  L2;
                               critical section 2;
                               turn: = 1;
                               remainder of cycle 2;
                               goto L2
                    end
            mend
       end".
```

2. アセンブリ言語または高級言語を使って，4.4.1節に提案したセマフォと同期基本命令とをエミュレートせよ．アセンブリ言語でエミュレートするときは，これらの基本命令をマクロで表わせ．何らかの高級言語で行なうときには，それらを手続きやサブプログラムとして組み立てよ．マクロ呼び出しや手続き呼び出しを出して，この章に述べた例についてセマフォや同期基本命令のテストをせよ．

3. 第一の p 操作が実行された後での事象の系列を述べよ．すべての仮定を述べよ．セマフォ(SYSSEM)は，第一の p 操作を実行するときに**1**であると仮定せよ．



4. この問題に答えるためには，次の定義が必要になる．**有向グラフ**(directed graph)は，空でない**節点**(nodes)の集合 N と**辺**(edges)の集合 E の対(N, E)である．E 内の各辺は，a, b が N の節点であるとき順序対 (a, b) で与えられる．(与えられた節点 a, b に対して，(a, b)という形式の辺が2つ以上 E の中にあってもよい)．辺(a, b)は，節点 a **から出ていて**(directed from)，節点 b **に向かう**(directed to)という．グラフは，節点 N の集合が2つの重ならない部分集合 $\prod$ と RHO とに分割できて，各辺の節点の一つは $\prod$ にもう一方は RHO に含まれているとき，これを**二部グラフ** (bipartite graph) という．

吸点 (sink) は，辺がそこから何も出ていない節点であり，**孤立点**(isolated node)は，そこから何も出ていないかそこへ何も向かっていないような節点である．路は，(a, b), (b, c), $\cdots$, (r, s)が辺のとき，少くとも2つの節点を含む**列** $(a, b, c, \cdots, r, s)$ である．路の最初と最後の節点が同一ならば，**輪**(cycle)という．

節点 a の**到達可能集合** (reachable set) とは，a から路が向かっているようなすべての節点 b の集合である．**結び目**(knot)とは，空でない節点の集合 K での各節点の到達可能集合が**ちょうど**集合 K になる場合である．

(a) 与えられた節点 a から与えられた節点 b に向かう辺が2個以上はないような二部グラフでは，E 内の辺の最大数は何か．

(b) 吸点，輪および結び目から成る二部グラフを図示せよ．

(c) グラフが結び目を持たない必要十分条件は，各節点が吸点かあるいは各節点から吸点へ向かう路があるということを証明せよ．

5. (R. Holt)**システム**を対$(\sum, \prod)$で定義する．ここで，$\sum$ は，状態の集合 $\{S, T, U, V, W, \cdots\}$ で，$\prod$ はタスクの集合 $\{T_1, T_2, T_3, \cdots\}$ である．

各タスクは，システムの状態からシステムの状態の部分集合への写像として定義される．図では，状態が $\{S, T, U, V\}$ でタスクが $\{T_1, T_2\}$ であるようなシステムを示している．このシステムでは，タスク T_1 は状態 S から $\{T, U\}$ へ写像する．これは，システムが状態 S にいるときにタスク T_1 が状態 T か U かどちらかへ変ることと解釈する．

タスク T_i が状態 U から状態 V を含む部分集合 $\sum$ に写像するとき，

$$U \xrightarrow{i} V$$

と書く．(“タスク T_i が U を V へ移す” と読む)．したがって，図では，

次のことがわかる．

$$S \xrightarrow{1} T,\ S \xrightarrow{1} U,\ T \xrightarrow{1} S,\ T \xrightarrow{2} S,\ T \xrightarrow{2} V.$$

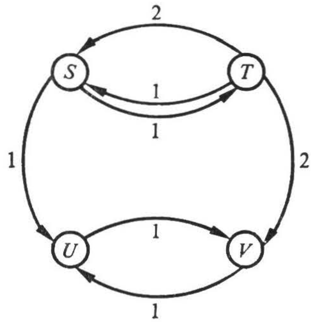
システムが次のような系列の0個以上の状態の変化をするとする．

$$S \xrightarrow{i} T,\ T \xrightarrow{j} U,\ \cdots,\ V \xrightarrow{x} W.$$

このとき，$S \xrightarrow{*} W$ と書く．

たとえば，図では，$S \xrightarrow{1} T$ かつ $T \xrightarrow{2} V$ だから，$S \xrightarrow{*} V$ である．T_i が一つの状態から別の状態へ移ることがもはや出来なくなったとき，T_i は**停止している**(blocked)という．T_i が，再び状態を変えられるようには決してならないとき，T_i は**デッドロックしている**(deadlocked)という．形式的には，

> タスク T_i は，$U \xrightarrow{i} V$ となる状態 V が存在しないとき，状態 U で**停止している．** タスク T_i は，$U \longrightarrow V$ なるすべての V に対して，T_i が V で停止しているときに，状態 U で，**デッドロックしている．**



(a)　図の中で，デッドロックしていないが停止しているタスク，およびその停止が生じている状態を識別せよ．

(b)　図の中で，デッドロックしているタスク，および，デッドロックの生じている状態を識別せよ．

(c)　次の命題は，定義から導けるか．

“タスク T_i が状態 U でデッドロックしていない必要十分条件は，$U \xrightarrow{*} V$ でタスク T_i が V で停止していないような状態 V が一つ存在することである”．

(d)　一個以上のタスクが状態 U でデッドロックしているならば，U はデ

ッドロック状態であるという． すべてのタスクが U でデッドロックしているならば，U は**全デッドロック**(total deadlock)の状態であるという．図において，もしあれば，デッドロック状態および全デッドロック状態を識別せよ．

(e) ある状態がどんなタスクによってもデッドロック状態に持ち込まれないとき，その状態は安全であるという．すなわち，

"状態 U は，$U \xrightarrow{*} V$ なるすべての V に対して，V がデッドロック状態でないとき，**安全**(safe)であるという"．

システムが一個以上安全な状態を含んでいれば，**安全**であるという．

$U \xrightarrow{*} V$ と仮定する．U がデッドロック状態のとき，V はデッドロック状態であることを証明せよ． また，U が安全な状態であるとき，V は安全な状態であることを証明せよ．

第5章

仮想記憶オペレーティング・システム

注　　意

この章では，次の問題に答えることが目的である．

1. なぜ我々は仮想空間が必要なのか．
2. 仮想空間の型とその能力とは何か．
3. 仮想空間を利用する多重プログラミングはどのように構造化するのか．

問題1.に答える際には，仮想空間は主として動的記憶割り付け，プログラム再配置の繰り返し，プログラムのオーバーレイ，プログラムやデータの実行時の成長などに関する問題を解くために設計していることを指摘しよう．仮想記憶計算機の開発者は，それぞれ，上記の問題を一つ以上解決しようと狙っていたので，いくつかの型の仮想空間が存在する．このために，仮想空間の型を分類し，例を使って説明しようと思う．その例には，CDC 3500, IBM 360/67, UNIVAC Spectra 70/46, IBM 370やHoneywell 645などがある．したがって，問題2.に答えることができる．

問題3.に答える際には，4つのオペレーティング・システム，すなわち，UNIVAC Spectra 70/46上のTSOS, IBM 360/67上のTSS, IBM 370上のOS/VS 2およびHoneywell 645上のMULTICSに共通な重要な特性をいくつか持つ“一般化した”多重プログラミング・オペレーティング・システムの概略を示そうと思う．これは，設計の研究という形で表示する．したがって，基本的なシステムの構成と詳細なプログラムの論理とを与える．一般化した多重プログラミング・システムを学ぶことにより，読者はシステムの概観をつかみ，システムのいくつかの要素間のインターフェースに習熟できることを期待する．

5.1 仮想空間と番地変換の概念

5.1.1 動的記憶割り当て

5.1.1.1 記憶利用の要求

主記憶に複数個のプログラムが並行的に存在する多重プログラミングの環境では，この記憶を効率良く利用したいという要求があることは明らかである．主記憶をもっと効率良く利用するために，システムは次のことをする**記憶割り当て機構**(memory allocation mechanism)を提供しなければならない．

1. 存在しているプログラムとそのデータとがもはや必要なくなったときには，他のプログラムやデータに与えるために，主記憶を使えるようにする．
2. 新しいプログラムやデータを，主記憶のどこか使える場所に置く．
3. 多重プログラミングの環境では新しいプログラムの到着は予想できないので，記憶の割り当てを動的に扱う．

1. を行なうために，システムは使用可能な主記憶の追跡をしなければならない．複数個の分離した使用可能記憶のブロックにより記憶の断片ができてしまうけれども，これらのブロックを追跡してゆくための仕事量は，管理できないことはない．難しいのは，新しいプログラムやデータを使用可能なブロックに動的に適合させることである．

5.1.1.2 プログラムの断片化と詰めの操作

主記憶が時刻 t_0 においてプログラム A, B, C および D によって占められている次のような例を考えよう．その時刻より後の時刻 t_1 においては，プログラム B と D は必要ないものとする．したがって，これらのプログラムによって占有されている主記憶は，使用可能としなければならない．

図 5.1 で，使用可能な記憶(斜線を引いた領域)は，2つの分離されたブロックを構成している．ここで，プログラム E を記憶に置くとしよう．プログラムはどちらのブロックよりも大きく使用可能な記憶の総和よりも小さければ，システムは，プログラム E を2つの部分に分けるか，または，主記憶にプログラムを置く前に使用可能なブロックを一つにまとめなければならない．

前者では，オペレーティング・システムがプログラムのいろいろな断片を管理しなければならない．記憶の断片化は時間と共に悪化するので，プログラム

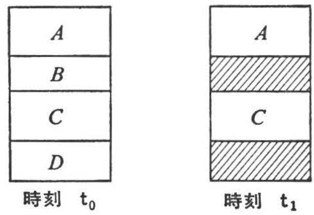


図 5.1　記憶の断片化

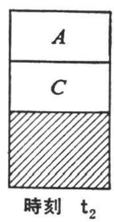


図 5.2　記憶の詰め操作

の断片を管理する仕事は大変なものになるであろう．

後者では，システムは，**詰め操作**(compacting operation)，すなわち，記憶の中のプログラムとデータとを動かして使用可能なブロックをまとめることをしなければならない．図 **5.1** に示した記憶に対する詰め操作の結果を，**図 5.2** に示す．

詰め操作に付随する問題は，次の通りである：

1. 詰め操作は主記憶から主記憶への操作であるから，実行中のプログラムにその操作の全期間にわたり割り込む必要がある†．すなわち，他のどれかのプログラムが追加の記憶を要求したので，現在のプログラムの

† たとえば，80ナノ秒の高速(キャッシュ)記憶と2マイクロ秒の主記憶をもったIBM 370/165では，N 文字を主記憶の一つの場所から別の場所へ移すMOVE命令は，大きな N に対しおよそ $t\,(= 0.56 + 0.24\,N)$ マイクロ秒かかる．すなわち，詰め操作をMOVE命令としてハードウェアで実現すると，一つのプログラムの詰め操作の間，N バイトのプログラムは，その実行を開始するために t マイクロ秒待たなければならない．

実行を中断しなければならない.

2. 動かされるプログラムとデータ(たとえば図 5.2 のプログラム C)は,番地に依存した情報(たとえば,同じプログラム内のエントリポイントの絶対番地)を含んでいるかもしれない. 番地と定数は記憶の中に数として格納してあるので,詰め操作で番地と定数とを区別することは不可能である. 番地を定数から区別するために使う情報は,再配置の情報というが,それは,言語翻訳系が作り——たとえば,アセンブラの内部再配置記号ディクショナリ——リンキングローダが絶対番地を計算する際に使う. しかし,第1章に示したリンキングローダはこの情報を退避しておかないので,これを使わないとプログラムの再配置を繰り返すために絶対番地の再調整をすることは不可能となる. そのようなリンキングローダは,通常,**静的リンキングローダ** (static linking loader) とよび,静的リンキングローダの行なう再配置は,**静的再配置**(static relocation)という. したがって,絶対番地に対する再配置の情報がもはや使えないので,詰めの操作で新しいプログラムの位置に関する絶対番地の更新をすることは不可能である.

詰めの操作は,リンキングローダによってプログラムを一度主記憶に配置した後で,それを再配置しようとする試みであることに注意しよう. プログラム再配置において詰めの操作が失敗するのは,一般に,プログラムが静的に再配置されると,再度再配置することができないことが原因となっている. このために,詰めの操作は,めったに使われない. したがって,もっと効率のよい動的記憶割り当ての機構を提供するために,プログラムがいったん再配置された後でそれを再配置できるような解を与えなければならない.

5.1.1.3 動的記憶割り当てとプログラム再配置の繰り返し

前述の問題に対する一つの解は,恐らく,最初のプログラムの番地とそれらに対応する新しい場所とを“覚える”ことのできる機能を提供する計算機システムの機能により決まる. 2つの異なった元の番地を同一の新しい場所に付随させたり,元の番地一つを2つの異なった新しい場所に対応づけることをしなければ,この機能は元の番地と新しい場所との一意的な対応を覚えることができる. 数学的には,このような関数は,1対1で単一値であるという.

もっと形式的にいうと,f は,プログラムのすべての番地の集合 V を定義域とし,記憶のすべての番地の集合 M を値域とする単一値で1対1の関数と

する．記号を使えば，次のように書ける．

$$f(a_i) = b_i,$$

ここに　$a_i \in V$　かつ　$b_i \in M$.

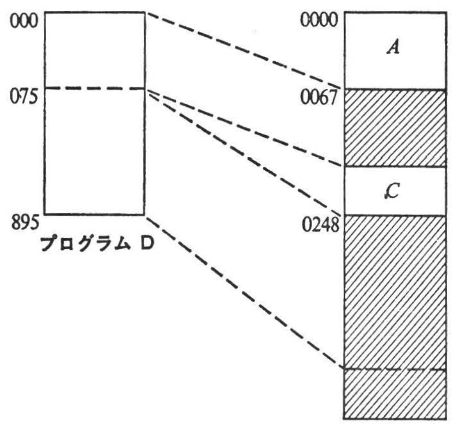
f が恒等関数 I ならば，

$$I(a_i) = b_i,$$

すなわち，すべての i に対し

$$a_i = b_i,$$

が成り立つ．言い換えれば，プログラム（もっと厳密にはそのロードモジュール）で使う番地は，記憶の場所そのものである．この場合，プログラムの再配置のために，プログラムの番地の再配置が必要になる．前節にも述べたよう



$f_D(x) = y$	
x	y
000	0067
001	0068
002	0069
⋮	⋮
074	0141
075	0248
076	0249
⋮	⋮
895	1068

図 5.3　プログラムの番地と記憶の番地の対応

に，これは不可能ではないけれども困難な問題である．

f が恒等関数でなければ，プログラムで使う番地は，プログラムが存在している番地と同一である必要はない．プログラムの番地と記憶の番地との間の正確な対応は，関数 f によって与えられる．

図 5.3 を見ると，次のことがわかる．

1. f_D が対応を管理しているので，記憶の場所に依らずプログラムの番地はやはり同じである．したがって，プログラムの再配置をする必要はない．
2. f_D は対応する記憶の場所を覚えているので，プログラムにとってはそれらの場所が分散していようと連続していようと問題ではない．したがって，記憶の詰めの操作は必要ない．
3. プログラムに対する動的記憶割り当てとは，使用可能な記憶を調べ関数 f_D を構成することである．

f_D が使えれば，プログラム C の再配置と詰め操作の両方が必要なくなる．しかし，システムは，上述の関数を構成しそれを使える機能を持っていなければならない．

実行中のプログラム D は，再配置時 t ごとに更新した関数 $f_D{}^t$ を与えられる限りは，動的かつ反復的に使用可能な空間どこにでも再配置することができる．したがって，計算機システムにおいて $f_D{}^t$ が実現できることは，詰め操作を除去し動的な割り当てと再配置の繰り返しとを実現するために役に立つ．

5.1.1.4 仮想番地，仮想空間および番地変換

プログラム D の命令を実行するとき，命令では特定の場所を参照するかもしれない．命令の番地部は，勿論，特定の場所の(通常のベースおよびインデックス修飾をもつ)番地をもっている．しかし，この番地は記憶の番地ではなくプログラムの番地である．プログラムの番地を使って，システムは関数 f_D に基づき記憶の番地を計算する．記憶の番地が計算されたならば，それを実際の参照をするために使う．記憶の番地を計算する過程を，**記憶変換**(memory translation)とよび，関数 f_D をプログラム D に付随した**変換関数**(translation function)または(関数が表形式になっていれば)**変換表**(translation table)という．プログラムの番地と記憶の番地の区別をするために，前者を**仮想番地**(virtual address)といいその番地付け空間を**仮想空間**(virtual space)というのに対し，後者を**物理番地**(physical address)といいその番地付け空間を**物理**

記憶(physical memory)という．この新しい術語を使えば，ユーザは仮想空間を与えられ，そのプログラムは仮想番地でコンパイルされ，アセンブルされるという．プログラムが物理記憶にロードされた後でその実行が行なわれる前には，そのプログラムに対する変換関数を作る．実行中に仮想空間を参照するごとに，この変換関数を介して仮想番地を物理番地に変換する．そして，この物理番地を使ってシステムは物理記憶の参照をする．ここで，情報が物理的に存在する記憶と，ユーザが参照を行なう情報のある記憶とは同一でなくてもよいことが明らかになった．仮想空間を採用している計算機システムを，**仮想記憶システム**(virtual memory system)という．

5.1.1.5　変換関数を構成する際の実用的考慮

上の議論によれば，システム内の各プログラムに対し，再配置時 t において変換関数 f^t が与えられる．これらのプログラムは動的かつ反復的に再配置されるので，変換関数 f^t に対する閉じた公式を作るのは難しい．そのかわりに，変換表を使い，f^{t_1} から f^{t_2} へ変換表を変更するときは，変換表 f^{t_1} のエントリを更新する．現在の仮想記憶計算機システムが変換表を採用するようになった主な理由は，この困難さのためである．これらの表に対し記憶を確保し変換の処理時間を減らすために，小さな表を使うことが望ましい．

図5.3を見ると，プログラム D が仮想空間と同じ大きさであれば，プログラムで使う各番地に対するエントリがなければならないので，変換表は仮想空間の大きさと同じだけ多くのエントリをもっていることがわかる．これは，明らかに実用的でない．仮想空間が物理記憶と同じ大きさだと仮定すれば，物理記憶全体を表の格納のために使わねばならずプログラムを入れる余地はなくなる．したがって，実用的な解が必要となる．

次のような物理記憶と仮想空間の構成を考えよう．

1. 物理記憶を語またはバイトの等しい単位に分割し，これらの単位を**ブロック**(block)とよぶ．
2. 仮想空間を**ページ**(page)とよぶ等しい単位に分割し，その大きさをブロックと同じにする．
3. 各プログラムに対し，プログラムのページとそのページが存在するブロックとの対応をつけるエントリをもつ変換表を構成する．

上述したことから，変換表のエントリの最大数は，仮想記憶の中の使用可能なページの数と等しくなる．

もっと厳密には，

2^m を仮想空間の大きさ，
2^n を物理記憶の大きさ，
2^k をブロックまたはページの大きさ，

としよう．ただし，語またはバイトの数で示す†．

したがって，仮想番地はビットの長さで，上位の$(m-k)$ビットをページの

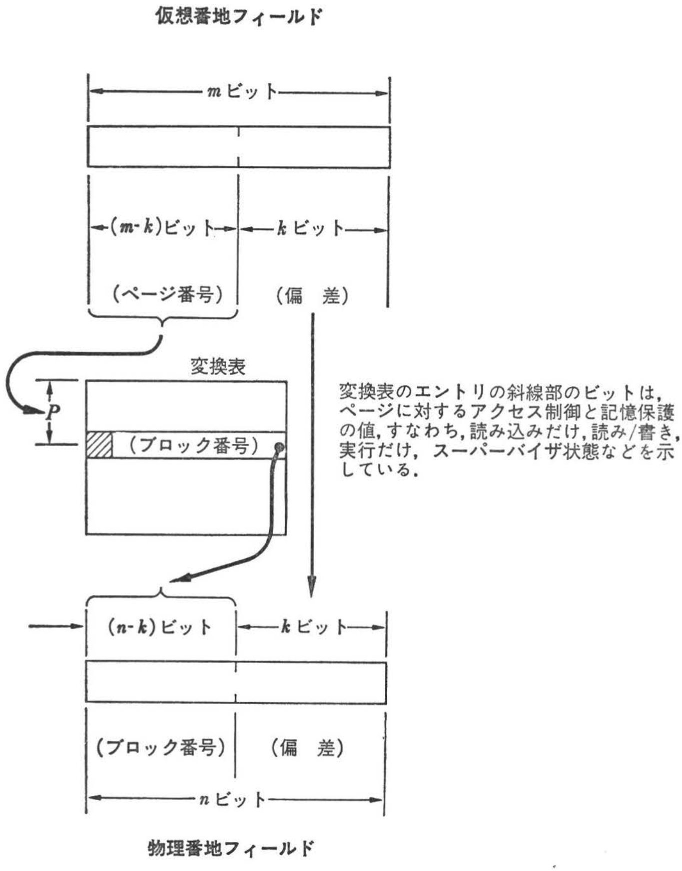


図 5.4　番地の変換

† 2進番地付け方式を使うと便利である．論理的には，計算機のハードウェアが同一の数システムである限りは，どのような数システムでも採用できる．

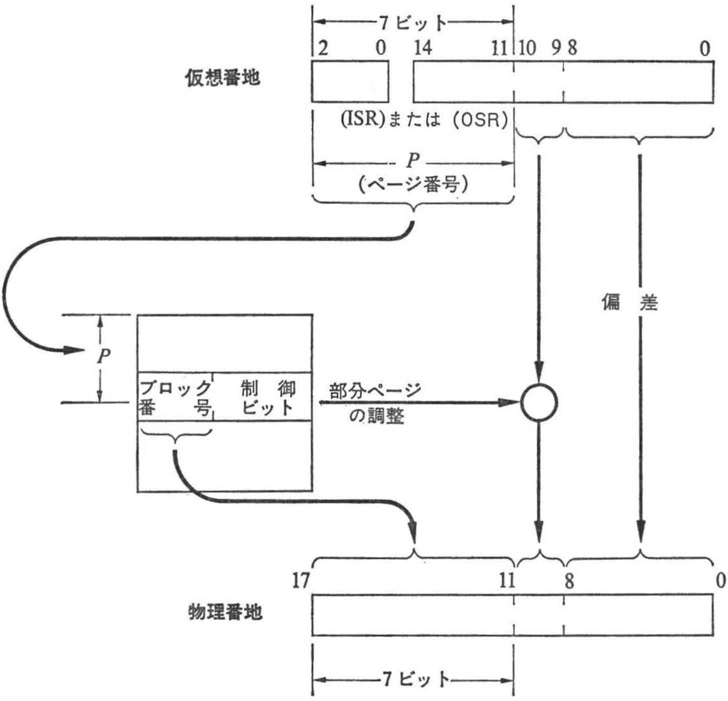
数を示すために使う．このため，これらの $(m-k)$ ビットを**ページ数** (page number)という．同様に，n ビットの物理記憶における上位の$(n-k)$ビットは，**ブロック数**(block number)とよぶ．3. から，一般に，変換表のエントリは，ページ数をブロック数に対応させることを学んだ．

図5.4より，ページ数を変換表の適切なエントリを捜すために使えることがわかる．このエントリは，(アクセス制御や記憶保護ビットのような)他の情報とともに，そのページに対するブロック数をもっている．そして，$(n-k)$ビットのブロック数を使って n ビットの物理番地を形成する．変換過程の間，k ビットの偏差はそのまま使う．この偏差は，与えられたブロックやページの中の片寄りを示す．

最も単純な仮想記憶計算機は，その仮想空間と物理記憶とが同じ大きさをもつもの，すなわち，$m=n$ となるものである．これは，CDC 3500 の標準モデル (**図5.5**参照) で実現され，CDC 3300 でもオプションとして使える．CDC 3500 の仮想空間の最大は，262,144 の 24 ビット語から成る．ページの大きさは，2,048 語である．したがって，128 ページ分ある．仮想空間の大きさと物理記憶とは同一であるから，ちょうど 128 ブロックの物理記憶がある．(**ページインデックス・ファイル**とよぶ) 変換表は，12 ビットの長さの 128 個のエントリから構成される．(**ページインデックス**とよぶ) 12 ビットのエントリは，7 ビットのブロック数とほかに 5 ビットの制御情報から成る．この制御情報は，(1) アクセスの制御たとえば読むだけ，読み/書き，あるいは番地付け不可能などを指定すること，(2) ページの一部の処理すなわち四分の一，二分の一，四分の三のページの処理をすることのために使う．もしも部分的なページを指定するならば，ページの大きさと番地とを与えなければならない．

変換表はただ一つしかないので，すべてのプログラムは同一の仮想空間内に存在しなければならない．さらに，すべての占有されているブロックの番号は変換表の中にあるので，ページを入れるためにあるブロックが使われているのかを示す**物理記憶利用表** (physical-memory-utilization map) をもつ必要はない．使用されていないブロックの番号は，変換表の中にはない．

(プログラムとデータの)ページは，勿論，言語翻訳系とそのサブシステムとが生成している．オペレーティング・システムは，リンキングローダの助けを借りて，変換表を介して使用可能なブロックを到着してくるページに割り当てる．ブロックが一杯になったときには，プログラム(すなわちそのページ)の実行が始まる．



CDC3500の実効ページ番号は，命令状態レジスタ（ISR）またはオペランド状態レジスタ(OSR)の3ビットの内容と，番地フィールドの上位から4ビットとによって作られる．これらのレジスタの内容は，オペレーティング・システムが管理し変更する．

図 5.5 CDC 3500 の仮想番地と記憶の変換

5.1.2 プログラム・オーバーレイのない仮想空間を目指して

CDC 3500 仮想記憶システムは，動的記憶割り当てとプログラムの再配置の繰り返しに関する問題を解決している．しかし，それは，(仮想)空間と(物理)記憶の利用率に関するような他の問題に対する解は与えていない．そのような問題の一つとして，第1章で物理記憶の利用と関連して論じてきた話題であるプログラムのオーバーレイに関するものがある．ここで，それをもう一度考えてみよう．プログラムが使用可能な空間よりも大きければ，オペレーティング・システムに特別な機能を与えて，ある番地をプログラムの異なった部分で異なった時間に使えるようにしなければならない(図 5.6 参照)．

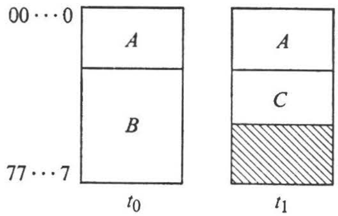


図 5.6

この例では，プログラム A は，時刻 t_0 で副プログラム B を，時刻 t_1 で C を呼び出すような主プログラムである．B と C のプログラムの論理は互に独立であるから，それらは異なった時刻に実行することができる．

プログラムのオーバーレイの技法は，次のことを必要とする．

1. オーバーレイされるプログラムの部分は，論理的には互に独立である．
2. 番地付けの衝突を避けるために，プログラムの結合と呼び出しとを注意深く行なわなければならない．
3. オペレーティング・システムは，リンキングローダがプログラムの部分を適切に再配置できるように，オーバーレイの構造を知らなければならない．

これは，明きらかにユーザ側とオペレーティング・システム側に大きな負担を課すことになる．

オーバーレイの問題を整理してみると，計算機内でオーバーレイ機能が必要になることは CDC 3500 に特定なものではなく，小さい仮想空間をもった仮想記憶システムや小さい実記憶をもった実記憶システムに一般的に見られることである．

5.1.2.1　線形仮想記憶システム

オーバーレイの問題を解決するために，仮想空間の基本的概念を利用しようという試みがなされてきた．一方で，仮想空間を十分大きくすると，プログラムのオーバーレイが必要なくなると期待される．また一方で，物理記憶を十分小さくすると，計算機の記憶のための追加の費用は大したものにならないと期

待できる．払うべき額は，記憶変換と補助記憶の費用と性能で決まるが，これについては，後ほど述べる．しばらくの間，仮想空間が物理記憶よりも大きいような仮想記憶計算機の例を示そう．

UNIVAC Spectra 70/46 計算機システムは，仮想記憶計算機である．その仮想空間は 4*K* バイトのページ 512 個から成り立ち，物理記憶は 4*K* バイトのブロック 64 個から構成される．すなわち，仮想空間は物理空間の 8 倍の大きさである．512 個のページがあるので，変換表(変換記憶とよぶ)には，高々 512 個のエントリができる．各エントリは，6 ビットのブロックの番号と 10 ビットの状態および制御の情報とからできている．後者は，ページが，

1. 修飾されたかどうか，
2. 存在しているかいないか，
3. オペレーティング・システムにだけ権利があるかどうか，
4. 読み出しだけか，読み出し/書き込みか，
5. 半ページか全ページか，
6. 半ページの場合，半ページの番地，

を示すために使われる．

仮想空間は実記憶の 8 倍以上大きいので，すべての仮想ページをブロックの中に入れることはできない．このため，補助記憶を用意する．補助記憶としては通常ドラムやディスクのような直接アクセス装置を一台以上使う．

記憶の変換に際しては，状態情報 2. が，参照しようとしているページが物理記憶にいないことを示しているかもしれない．このような場合，ページはどこのブロックにもなく，ページフォールトが生ずる，**ページフォールト**(page fault) はハードウェアの割り込みで，これにより，オペレーティング・システムは**ページング処理**(paging process)を開始し，補助記憶から目的のページを物理記憶の中に持ってきて使えるブロックの中に置く．ページが持って来られる間に，オペレーティング・システムは現在の変換表を他のプログラムの変換表と置き換えることにより，他のプログラムへ制御を渡してもよい．オペレーティング・システムのページングのルーチンは，補助記憶(たとえばドラム)の管理とページの格納および検索に責任をもっている．前述した詰め操作と異なって，ページングは，主記憶から主記憶への操作ではない．これは，主として，入出力操作である．CPU の制御を別のプログラムに与えてやれば，入出力と計算の操作の重ね合せが期待できる．さらに，ページングによって割り込まれるプログラムは，実行中のどれか別のプログラムではなく追加の空間を要

求しているプログラムそのものである．

2百万バイトあるUNIVACの仮想空間は，動的記憶割り当てやプログラム再配置の繰り返し問題を解決しようとするだけでなく，プログラムオーバーレイの問題も解決する積りである．潜在能力としては，各ユーザには，512ページ——2百万バイトの仮想空間を与えることができる．各ユーザのプログラムのページは，64ブロックの物理記憶を競い合う．実際は，次の理由により，そのようなことにはならない．

1. 補助記憶が小さい．すなわち，ドラムは800ページ分の空間しか持っていない．
2. オペレーティング・システムは，自分自身のために各変換表の最後

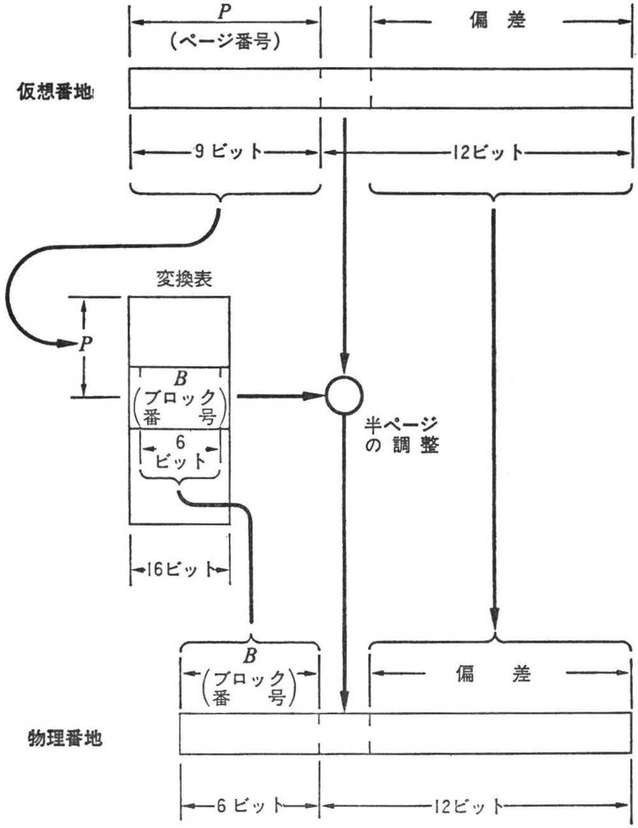


図 5.7　Univac Spectra 70/46 の仮想番地と記憶の変換

256 のエントリを確保しているという制限がある.

図 5.5 と図 5.7 とを比較すると，両者の番地変換過程は似ていることがわかる．主な違いは，図 5.7 ではページ番号のビットはブロック番号のビットよりも大きいのに対して，図 5.5 ではこれら 2 つの番号は同一のビット数をもつことである．CDC 3500 と UNIVAC Spectra 70/46 の仮想記憶は，両方とも線形方式，すなわち，(00･･･0)から(77･･･7)であるから，これを，**線形仮想記憶システム**(linear virtual memory system)という．これらの記憶変換では，変換表の一次元(あるいは一ステップ)探索という特徴がある．このため，これらを，しばしば，**一次元仮想記憶システム**(one-dimensional virtual memory system)ともいう．仮想空間が物理記憶よりも大きいときには，次のような要求がつけ加えられる．

1. ドラムやディスクのような補助記憶の必要性.
2. オペレーティング・システムが補助記憶からページを検索し，また(新しいかあるいは更新した)ページをそこへ格納する能力．検索の過程を**ページングイン操作**(paging-in operation)，格納の過程を**ページングアウト操作**(paging-out operation)という．これらを合わせて，**ページング**と呼ぶ.
3. **補助記憶使用表**(backing-storage-utilization map)および**物理記憶使用表**(physical-memory-utilization map)の管理．前者は，ページに対する補助記憶の記憶単位(たとえば，セクタやトラック)の使用を保持している．後者は，ページングに対する主記憶のブロックの使用を覚えている．このような使い方をするので，これらは，また，簡単に，**仮想空間表**(virtual space map)および**物理記憶表**(physical memory map)ともいう.

物理記憶使用表を使えば，仮想記憶システムは複数個の変換表(したがって複数個の仮想空間)を実現することができる．異なった仮想空間のページがブロックに入っているので，この表により与えられたブロックに対してそのページが属している空間を示すことができる．使用できるブロックが少ないときには，物理記憶からいくつかのページを“追い出さねばならない”かもしれない．この場合，追い出されたページの仮想空間を識別しなければならない．一旦識別されれば，それに対応する変換表をさがし，これらのページに対する表のエントリを変更することができる.

ページングの操作，表の管理，多重仮想空間については，後の節でもっと完

全に述べる積りである．ここでは，実記憶よりも大きな仮想空間をもつ線形仮想空間の潜在的能力と，システムの価格を大幅に増大しその性能を低下させる精巧なシステムの操作のオーバーヘッドについて指摘するだけにとどめる．大きな仮想空間を使うという理由づけは，勿論，補助記憶の方が物理主記憶よりも低価格であること，一つのプログラムに対するページングの処理は大部分他のプログラムの実行と重ね合ってしまうこと，表類の管理がそんなに大変ではないことなどが期待できるからである．さらに，これらのシステムでは，プログラムの再配置の繰り返し，動的な記憶の割り当て，オーバーレイ構造の除去などができる．

5.1.2.2　線形セグメント化仮想記憶システム

IBM 360/67 仮想記憶計算機システムは．最大 2^{32} バイト(40億バイト以上)の仮想空間をもつ．ページの大きさは 4K バイトなので，100万個以上($2^{32} \div 2^{12} = 2^{20}$) ページがある．前にも述べたように，各ページに対して変換表に1エントリづつあり，ページ番号とブロック番号との対応をつけている．したがって，変換表の最大の大きさは 2^{20} である．変換表の 2^{20} 個のエントリを一次元探索すると，20ビットのインデックスレジスタが必要になる．さらに，線形の表は，滅茶苦茶に大きくなる．

そこで，二次元探索(2ステップの表探索)を開発して，大きなインデックスレジスタおよび長い線形の表の必要性を除去する．**図5.8**を見ると，仮想空間のページを，**セグメント**(segment)に分割している．2^{12} セグメントがあり，各セグメントは 2^8 ページから成り立つ．**セグメント表**(segment table)とよぶ表には，各エントリに対して**ページ表** (page table) とよぶ別の表の先頭がある．セグメント表内のエントリの最大数は，2^{12} である．すなわち，最大 2^{12} のページ表がある．各ページ表は，2^8 個のエントリをもち，これがページ番号と対応するブロック番号との関係を与えている．したがって，2^{20} 個のページ表のエントリの代りに，4352 $(2^{12} + 2^8)$ のエントリがあればよい．

番地の変換は，仮想番地の先頭から12ビット，すなわち**セグメント番号**(segment number)をセグメント表により解読することによって行なう．セグメント表のエントリより，ページ表の先頭がわかる．仮想番地の次の8ビットを，ページ番号として使う．ページ表内の適切なエントリを見つけると，ページ番号がわかる．ブロック番号と偏差とから，物理記憶番地を構成する．

このように，番地変換は二次元探索である．このため，IBM 360/67 型の仮想空間は，**二次元仮想記憶システム**(two-dimensional virtual memory sys-

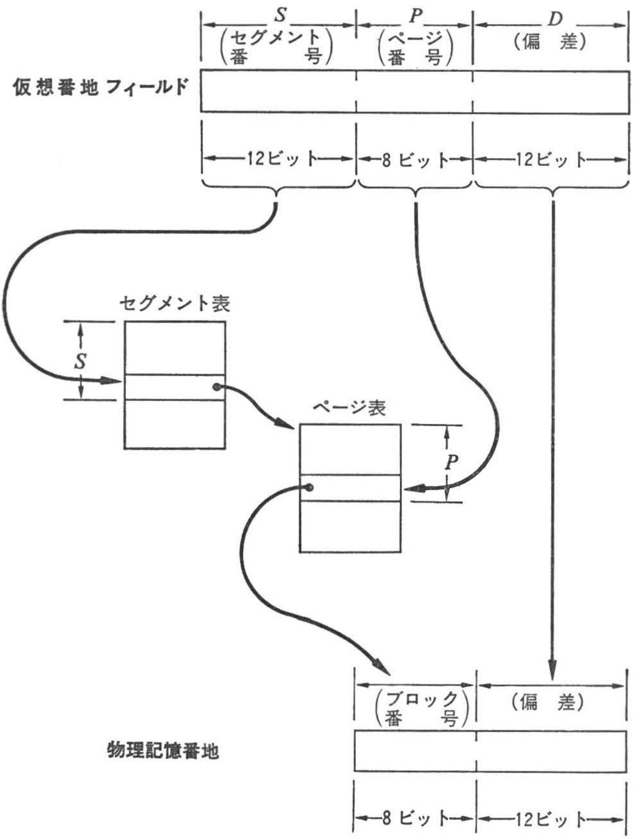


図 5.8 IBM 360/67 の仮想番地と記憶の変換

tem)とよぶ．仮想空間のセグメンテーションとページングをするので，このシステムは，**線形セグメント化仮想記憶システム** (linear segmented virtual memory system)と名付けることもできる．

IBM 370 は，線形セグメント化仮想記憶システムを改良している．IBM 370 では，望んでいるセグメントとページの大きさに応じて，複数個の仮想空間の構成を選べる．2つの異なったセグメントの大きさ(64K と 1M バイト)と2つの異なったページの大きさ(2K と 4K バイト)があるので，4つの構成が可能である．さらに，ページ内の偏差はページの大きさによって影響を受ける．4つの仮想番地と2つの物理番地の形式を，図 5.9 に示す．

この4つの構成のうちどれを選ぼうとも，仮想番地の最初の8ビットは，

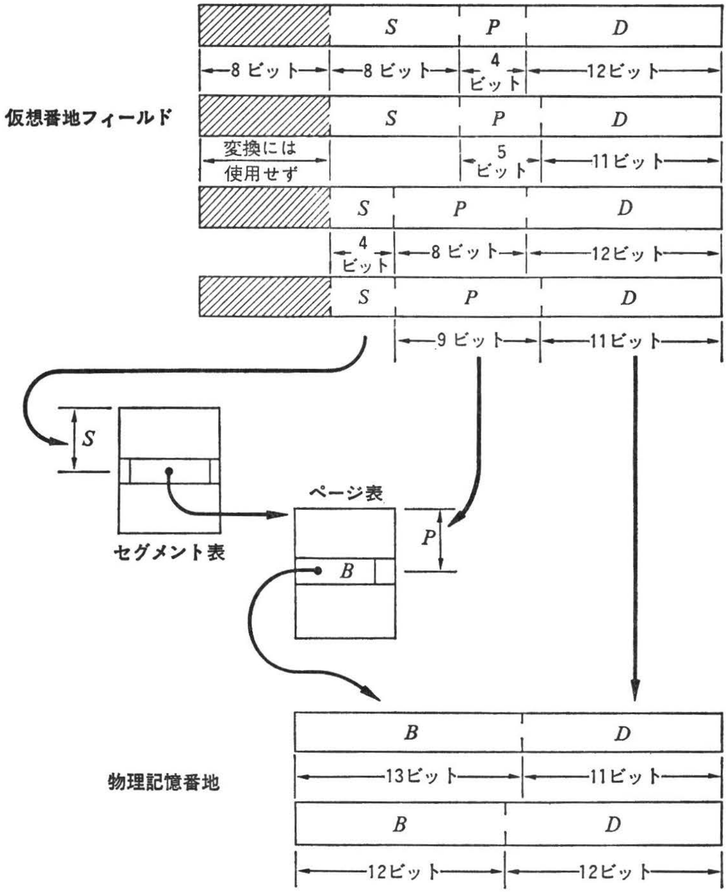


図 5.9　IBM 370 の仮想番地と記憶の変換

(現在のところ)番地変換には使わない．したがって，残りの24ビットだけが仮想番地として使われる．物理番地も24ビットであることに注意しよう．仮想および物理番地が両方とも同じビット数をもっているので，仮想空間および物理記憶の大きさは，CDC 3500 の場合†と同じことになる．CDC 3500 と IBM 370 の仮想記憶システムの違いは，CDC 3500 では仮想空間は唯一つしかないのに，IBM 370 では一個以上の仮想空間が存在できることである．IBM 370 が実現できる仮想空間の数は，UNIVAC Spectra 70/46 の場合と同じように

† IBM 370 のすべてが，2^{24} バイトの物理記憶の容量を備えているわけではない．

補助記憶の規模によって決まるのであり，番地変換機能によって制限されるわけではない．もっと詳細に述べると，IBM 370 仮想記憶システムは，2^{24} バイトの単一の仮想空間かまたはそれぞれ 2^{24} バイトの大きさをもった複数個の仮想空間を提供するように構成できる．それにも拘わらず，物理記憶は，最大限 2^{24} バイトの大きさをもつ．

大きな仮想空間を実現するために，2つの方法があるのは興味深い．一つの方法は，物理番地よりも仮想番地に多くのビットを許すことである．仮想番地のフィールドの方が物理番地のフィールドよりも長いので，仮想空間は物理空間よりも大きい．もう一つの方法は，仮想空間を複数個許すことである．この場合，仮想番地が物理番地と同じビット数をもっていても，空間が多数あれば，実質的に記憶の大きさを多数倍したことになる．後者が，IBM 370 の方向である．さらに，仮想番地が物理番地よりも長くて仮想空間の多重度が1よりも大きければ，UNIVAC Spectra 70/46 や IBM 360/67 の場合のように記憶は両方向に対して拡大される．図 5.9 をもう一度考えてみよう．IBM 370 仮想番地で使用していない部分（8 ビット）を，個々の仮想空間の将来の拡張のために使うことができる†．

要約すると，大きな線形セグメント化仮想記憶システム――IBM 360/67, UNIVAC 70/46, IBM 370 など――は，(1) 動的な記憶割り当て，(2) プログラム再配置の繰り返し，(3) プログラムのオーバーレイ，の問題を解決できる．

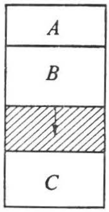
5.1.3 まばらに使用される線形セグメント化仮想空間

5.1.3.1 セグメンテーションと単層記憶の概念

次の例を考えてみよう．A, B, C は3つの実行中のプログラムである．プログラム B が実行中に大きくなると考えるならば，B に対して追加の空間（斜線の領域）を確保してその成長が確保した領域内におさまるようにすることができる．そうでなければ，プログラム B は，オペレーティング・システムに動的に空間を要求しなければならない．

前に述べたように，記憶管理は，オペレーティング・システムを大幅に複雑にする．プログラマがプログラムの成長のために空間を確保しなければならな

† IBM 370 の仮想空間の大きさを拡大する別の方法は，新しい仮想空間の単位を作ることである．使用していない 8 ビットの番地フィールドを，新しい単位の番号を示すために使える．セグメント表をさがすために，新しい単位の表を参照する．新しい単位の番号は，セグメント表の先頭を与えるエントリを見つけるための偏差として使う．以後の変換過程は，図 5.9 に示したものと同じである．



いのは，仮想空間が線形番地空間として使われているからである．線形番地空間では，一つのプログラムが成長すると隣りのプログラムに影響を与えるかもしれない．追加の空間を確保することをあらかじめしてあっても，プログラムはそれに割り当てられた空間をはみ出して成長するかもしれない．プログラムの成長が予測できない場合には，特にそれが生ずる．一つの例が ALGOL や PL/I における動的配列の状態であり，そこでは，配列の“次元”は固定しているわけではない†．したがって，その成長は通常終りがない．別の例は，ファイルの処理である．ファイルはデータの集まりである(実際，IBM では**データセット**(data set)という術語でよぶ)から，やはり成長の問題がある．

プログラムとデータの成長の問題に対する一つの解は，線形セグメント化仮想空間を**まばらに**(sparsely)に使うことである．たとえば，プログラムやデータに常にセグメントを割り当てるならば，そのセグメント内でプログラムやデータが成長しても，そのセグメントに対するページ表にエントリを一つ追加するだけで良い．充分の数のプログラムやデータセットを実現できるようにセグメントがあり，各セグメントはたくさんのページをもっていてプログラムやデータセットがセグメントの大きさを越えることが殆んどないならば，成長に関する問題は解決される．

別の利益もある．たとえば，再入可能型プログラムの場合，プログラムの純手続きに一つのセグメントを割り当て，そのデータに別のセグメントを割り当てる．したがって，変更されないプログラムの論理と，変更されるデータとが分離される．さらに，純手続き間の結合とそれらに付随したデータの管理とが，セグメント間の通信として(必ずしももっと効率良くはないが)もっと明確に処理できる．ファイル処理の場合，セグメントをファイルとして使うように

† **訳注** 配列の次元は，実行時に，手続きのパラメータ，全域変数など外側から与えられた環境の状態によって決まる．

訓練しても良い．ファイルに対する参照は，セグメントへの参照となる．それゆえ，ファイルのデータ処理は，ユーザからは仮想空間におけるデータの転送と見ることができる．データ処理をテープやディスクのような二次記憶に対する入出力操作と見る伝統的な方法は，すぐに消えてしまう．ユーザに関する限り，自分のファイル，プログラム，データ，共通のサブルーチンは，すべてその仮想空間に格納されている．ドラム，ディスク，テープやその他の補助記憶の多数の層は，ユーザには“見えない”．このように仮想記憶を使用する概念を，**単層記憶概念**(one-level storage concept)という．セグメントを強調する意味は，通常のセグメンテーションとはかなり変ってきていることに注意しよう．ここでは，セグメンテーションは仮想空間の単なるもう一つの次元というだけではなく，プログラムやデータを仮想空間で扱うときの原則となるものである．

セグメントを使ってプログラムの成長を扱うことと単層記憶の概念を使うこととは，この章の終りのあとがきで示すようにどこか別々の起源から出発しているけれども，仮想記憶システムに対し良い原則に基づいたセグメンテーションと単層記憶の概念とを統一化し一般化することが，MIT で示されてきた．MIT の MULTICS†は，2^{14}(16K)セグメントと各セグメントが256個の10K語のページから成る Honeywell 645 仮想記憶システムの上に実現された．すなわち，ユーザは，概念的には2^{14}個のプログラムやファイルを書いたり作ったりでき，各プログラムやファイルは2560K語まで成長することができる．Honeywell 645 における番地変換は，IBM 360/67 のものと似ている．すなわち，二次元探索である．これについては，再度述べることはしない．

5.1.3.2 まばらに使用される線形セグメント化仮想記憶システム

MULTICS の Honeywell 645 仮想記憶システムは，**まばらに使用される線形セグメント化仮想記憶システム** (sparsely used linear segmented virtual memory system)とよばれる．これは恐らく最も野心的なシステムであり，システムプログラミングの中であらわれる最も顕著な5個の問題を解決する積りである．その問題とは，

1. 動的な記憶割り当て，
2. プログラム再配置の繰り返し，
3. プログラムオーバーレイ，

† 訳注 Multiplexed Information and Computing Service の略．MIT の MAC プロジェクトで，本格的な時分割システムとして開発された．

4. 実行時のプログラムおよびデータの成長，
5. 単層記憶の処理，

である．

この時点で，将来の仮想記憶システムはすべて問題1.から3.に加えて問題4.と5.との解決を目指すべきかどうかを問いただしたい．データセットが小さく一時的なもの（たとえば，プログラム呼び出しに対するパラメータリストなどで，恒久ファイルでないもの）である限りは，問題4.はIBM 370のような線形セグメント化仮想記憶システムでも困難なく解決できる．しかし，問題5.の解に関しては，かなり異なった意見がある．セグメント化した仮想記憶システムにおいて単層記憶の概念を実現するためには，次に示すような取り決め，機能および技術が使えなければならない．

1. すべてのセグメントは，記号名を持っていなければならない．（ファイルは，ファイル名を持っていることに注意しよう）．
2. "伝統的"なファイルシステムは，セグメント管理システムと置き換えなければならない．カタログされたファイル名は，カタログされたセグメント名と置き換えなければならない．（後の節におけるカタログとファイルシステムについての議論を参照せよ）．
3. セグメントの規模は，大きなファイルを収容できるように充分大きくなければならない．（現在のところ，Honeywell 645における2560K語のセグメントは適当ではない）．
4. 有効なページングアルゴリズムと効率の良いセグメント間の通信機構によって，ファイルのデータ処理を推進しなければならない．小数のプログラムやデータセットだけが関与し，プログラムやデータセットへの参照が局所化され頻繁でないようなプログラムの準備や虫取りといった会話型プログラミングとは異なり，データ処理の環境に関与するファイルの数は大きい．さらに，データの量は巨大で，データへの参照は頻繁であり，参照箇所はばらまかれている．もっと有効なアルゴリズムを使わないと，ページングの量は過度になってしまう．さらに，頻繁なセグメント間の通信（すなわち，セグメントの境界を越える記号による飛び越し結合——branch and link)は，時間のかかるものになり得る．
5. 記憶の技術が充分発達して，いろいろの補助記憶を使った記憶の階層の構成の問題，価格と使用期間に基づいた階層内のデータの自動的移動の問題が解決できなければならない．

現在のところ，項目 4. については良くわかってはいない．さらに，記憶の階層の技術は，まだ先が長い．セグメント管理者が生成し管理しているファイルと，伝統的なファイルシステムが生成し管理しているファイルとの互換性の無さは，単層記憶への動きを更に複雑なものにする．簡単に言えば，まばらに使用する線形セグメント化仮想記憶システムを導入しなければならないとしたら，現存するファイルすべてに対し何をしたら良いであろうか．IBM 370 の線形セグメント化記憶システムは，用心深い方法を採用した．伝統的なファイルシステムを保持しておき，単層記憶の概念は実現していない．その代りに，仮想記憶アクセス法(VSAM: Virtual Storage Access Method)を提供し，そこでは，データセットの要求時に二次記憶に格納してあるデータセットを仮想空間に写像することができる(5.2.2.4 節の仮想アクセス法の議論を参照せよ)．一度仮想空間に写像されれば，データセットを直接参照することができる．仮想記憶計算機の異なった開発者が上述の問題の幾つかを解決しようと努めているので，仮想空間の構成や実現にはいろいろの型がある．この目的のために，CDC 3500, UNIVAC Spectra 70/46, IBM 360/67, IBM 370, および，MULTICS-Honeywell 645 などの型を例として説明してきた．

5.2 一般化した多重プログラミング用オペレーティング・システム

仮想空間を使う一般化した多重プログラミング用オペレーティング・システムを提示する目的は，次の 2 つである．

1. いくつかの多重プログラミング用オペレーティング・システムに共通な性格を要約して，これらのシステムの概観を読者が得られるようにする．
2. 一般化した多重プログラミング用オペレーティング・システムの概略を示し，そこから重要な概念を詳細にかつ特殊化して論ずる．

このシステムは，現存する 4 つの仮想記憶システム，すなわち，UNIVAC Spectra 70/46 上の TSOS, IBM 360/67 上の TSS, Honeywell 645 上の MULTICS, および IBM 370 上の OS/VS2 から抽象化して得られたものである．物理的資源の割り当てに最も大きな影響を与えそうなシステムの部分について，この設計では詳細に論じる．したがって，次のような特徴については，はっきりと示している．すなわち，多重プログラミング，要求ページング，多重タスキング，入出力機器に対する多重路，動的ローディング，階層的なファイルカタログ，および仮想アクセス法がそれらである．タスクの開始部，ディス

パッチャやページングのアルゴリズムのように，システムごとに大きく変るようなアルゴリズムについては，何も詳しいことは述べない．しかし，システム内におけるそれらの位置と機能については，はっきりさせておく．

5.2.1 ハードウェアへの要求

いま論じている多重プログラミングシステムは，単一処理装置のシステムである．この議論から得られる結果は，多重処理装置システムに拡張してもよい．動的な記憶割り当ての主要な方法は，要求ページングである．4つのハードウェアに関する節は，次の事項を扱う．

1. 処理装置(中央処理装置あるいはCPUともいう)，
2. 物理的記憶(主記憶あるいはコア記憶ともいう)，
3. 補助記憶(バックアップまたはスワッピング記憶ともいう)，
4. 入出力チャネルと二次記憶機器(外部機器ともいう)．

5.2.1.1 処　理　装　置

タスクが処理装置の制御を得るためには，次のような操作をしなければならない．

1. タスクのプログラムが制御している汎用レジスタを，タスクに対して適正な値に設定しなければならない．
2. タスクの特権や状態を決定する情報を含むようなCPUの他のレジスタの設定をしなければならない．
3. 仮想番地を物理番地に変換するハードウェアの機構を生かさなければならない．この操作により，MULTICS-Honeywell 645におけるディスクリプタ・ベース・レジスタ(DBR)やIBM 370の制御レジスタのようなすべてに重要なレジスタに，タスクの変換表のポインタを回復しておく．あきらかに，これは一つの仮想空間を他のものから分離するためのレジスタである．さらに，また，このレジスタは，現在のタスクに関連のあるいくつかのタスクが同一の仮想空間にアクセスできるようにするものである．そのうえ，連想記憶レジスタや高速バッファレジスタを利用する並行的な変換機器が何か使えるのならば，それも生かしておかなければならない．これらのレジスタは少ししかなく，まだブロックに残っていて一番最近ページに割り当てられたブロックの番号だけを含んでいる．これらのレジスタを使うと，仮想番地がこのレジスタに入って

いるブロック番号のページをちょうど参照しているならば，変換の過程を短縮することができる．そうでなければ，ページ表(およびセグメント表)を介した典型的な一次元(二次元)探索をしなければならない．

4. プログラムカウンタは，次に実行すべき命令の仮想番地をもっているので，タスク内で実行すべき次の番地に設定しなければならない．この操作により，実際にタスクへの転移が起る．

タスクが処理装置を放棄するときには，上記の情報はすべて退避しなければならない．したがって，処理装置上のタスクの切り換えには，ある程度のオーバーヘッドが含まれる．

計算に必要な処理装置の時間は，処理装置が使っている記憶装置との間のデータのやり取りの量に比例する．一般に，外部機器は回転しており，各バイトが転送される時間はばらつきが大きい．したがって，必要なだけの記憶サイクルを使うであろう．そこで，処理装置は，チャネルによって使われる記憶サイクルの量だけ遅れてしまうであろう．

5.2.1.2 物理的記憶

物理的記憶のブロックには，常に，次のものがある．

1. 処理装置が使っているか，あるいは，処理装置によって実行されるのを待っているプログラムやデータの領域．
2. 入出力装置やチャネルを待っているか，あるいは，チャネルに内容を転送したりそれから受け取っているような入出力バッファ領域．
3. 現在はユーザはいないけれども，速い応答を必要としたりあるいはその使用の期待値が大きいなどの理由で，要求があったときに使用できるようにしておくようなプログラム．
4. 入出力操作や処理を中断させている他の操作の終了を待っているタスクのプログラムやデータの領域．
5. 何の役にも立たないもの．したがって，これらのブロックは，割り当てに使ってよい．

物理的記憶は，別々にアクセスできて，一つのバンクのチャネルの操作が別のバンクの処理に影響しないようにバンクやモジュールに構成されているかもしれず，それによって資源利用の問題に新しい次元を追加することになるかもしれないけれども，そのような記憶システムは直接考えない．以後の節で論ずることは，n 個の記憶バンクをもつような記憶システムに適用するように拡張

できる．たとえば，ページングの操作が行なわれるとき，一般化したシステム内で n 個の記憶バンクのどれかでページングの処理をしなければならないか，またもしそうであれば，どのバンクにそれを施すべきかを決定しなければならない．バンクが相互にまたはシステムの他の部分と相互作用を行なう規則については，簡単に一般化することはできない．

5.2.1.3 補 助 記 憶

補助記憶は，ドラム，固定ヘッドディスクまたは大容量コアである．大容量のコア記憶を主記憶のバックアップとして採用しているシステムに関する最近の報告では，特に要求ページングシステムにおいては，全体の性能が飛躍的に向上していることを示している．大部分の多重プログラミングシステムでは，もっと費用の安いドラムやディスクをまだ利用している．

大規模な大容量コア記憶は，全く乱アクセス機器であると仮定する．すなわち，どこの場所でも，ドラムやディスク装置と比較して非常に小さいアクセス時間でアクセスすることができる．ドラム（ディスク）は，ページが n 個の異なったセクタ（トラック）に分散されており，そこから，一つの操作が回転ごとに一回づつ周期的に行なわれる．一時には，ただ一つのページが一つのセクタ（トラック）から読み込まれる．したがって，一つのセクタ（ページ）に対して2つのページの要求が同時に出されると，一方は，平均1/2の回転周期になる遅れ時間の後で，もう一方は，一回転以上待たなければならない．

物理的記憶は，一つのページング用チャネルでサービスされる．これは，多数の記憶バンクがある場合には，ある程度の努力をすれば多数のページング用チャネルに拡張することができる．

ページング用チャネル自身は，任意の時点で次の動作をしている．

1. 要求ページの入力で，最高優先度の動作．これは，物理記憶内のタスクがその処理を続けることができるように，一つのページを物理記憶内に持ってくるところである．
2. 前ページング．これは，タスクが処理装置に与えられる前に，タスクのページを物理記憶の中に持ってくるところである．
3. ページ出力．これは，ページを物理記憶から補助記憶へ放り出しているところである．

5.2.1.4 入出力チャネルと機器

システムの入出力の部分では，多数のデータチャネルを備え，いろいろの大

きさと速度の直接アクセス機器のサービスをする．チャネルに対する機器の割り当ては，一意的ではない．一般に，一つの入出力機器に対して異なったチャネルを介してアクセスできる．

入出力機器は，ユーザの端末とのインターフェースを提供する通信装置も含んでいる．端末はチャネルに比べて非常に低速であるから，すべての端末が一つの通信装置に多重化されており，その装置はまた一つのチャネルに他の低速の装置と一緒に多重化されている．ユーザの端末との通信は，端末が低速なので，通信装置に対して比較的定常的な負荷を与えることになる．端末が低速でゆったりしているために生ずる通信用のチャネルや機器の使用の低減は，それをシステムに対する内部的な修正だけでは防ぐことは出来ないので，許容するようにしなければならない．したがって，このチャネルのスケジューリングは，全体の資源割り当て問題にとっては重要な部分にはならないであろう．

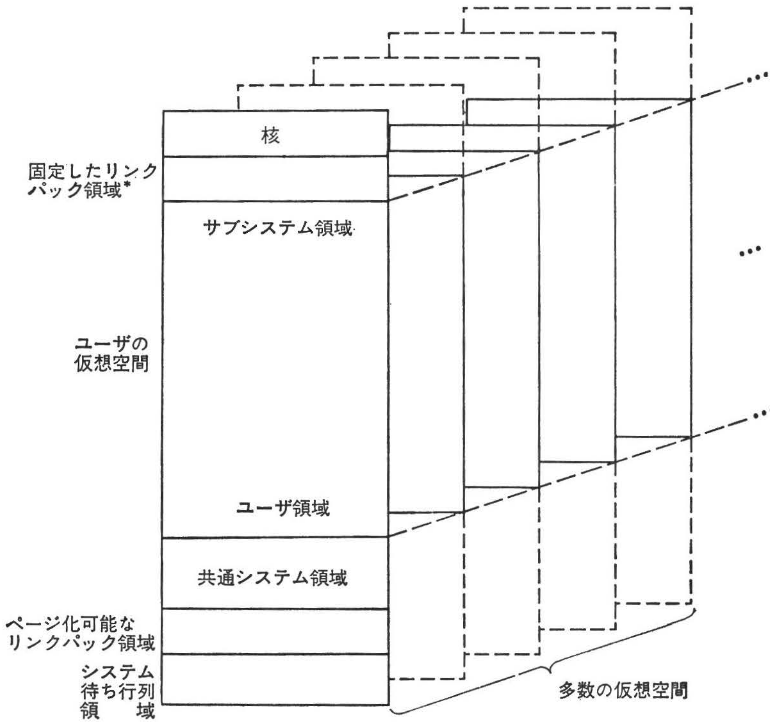
乱アクセスの入出力機器は，その転送時間はページングの機器と同じオーダであるけれども，それほど高速ではない．それらは，ファイル，ファイルのカタログ，そのほかのオフラインおよびオンラインの記憶のために専用的に使う．

5.2.2 ソフトウェアシステムの構成

仮想記憶システムとして，システムの仮想空間は，次のような制限を持つように構成される．

1. 各仮想空間の中で同一のセグメント番号をもついくつかのセグメント(通常，低番地あるいは高番地の端，またはその両方)が，システムに恒常的に割り当てられている．すなわち，ユーザのタスクは，これらのセグメントを直接使うことはできないであろう．これらのセグメントを使う必要があるときには，これらのセグメントに対するユーザの要求はスーパーバイザコール(SVC)のような割り込みとなる．
2. これらのセグメントのいくつかは，物理記憶内に恒常的に保持しておく．

仮想空間の一部をシステムが使用するために恒常的に割り当てることは，セグメント表のエントリを制御すれば実現できる．オペレーティング・システムがユーザのタスクに対するセグメント表を作成するので，セグメント表内のいくつかのエントリを自分自身が使用するために確保する．たとえば，UNIVAC Spectra 70/46では，各変換表の終りの256エントリは，TSOSオペレーティ



* リンクパック領域は，共用可能な再入可能型プログラムの領域である（4章参照）

図 5.10　IBM 370 OS/VS 2-2 仮想空間の配置

ング・システムのために確保してある．IBM 370 では，図 5.10 に示すように OS/VS 2-2 もやはりいくつかのセグメントを自分の使用のために確保している．さらに，これらのセグメントのある種のページは，いつでも物理記憶のブロックに入れておく．たとえば，そのようなページが，図 5.10 に示したように，オペレーティング・システムの核を構成している．

オペレーティング・システムは，システムのいくつかのハードウェア資源を，システム内で動いているタスクに割り当てる．別の見方をすれば，オペレーティング・システムは，ハードウェア資源に対してタスクをスケジュールしている．各タスクに対して，オペレーティング・システムは，タスクに関するすべての情報に対する**タスク制御ブロック**(task control block；略して **TCB**)を維持している．各タスクは，また，そのタスクに割り当てられた仮想空間の

仮想番地から主記憶のブロックの物理番地への写像によって特徴づけられる．この写像は，タスクのセグメントおよびページ表によって与えられる．

5.2.2.1　オペレーティング・システムの常駐部

割り込みや SVC が生ずると，CPU の割り込み処理のハードウェアが，割り込みに関する情報——たとえば，割り込みが生じた場所——と割り込みの型を識別する番号とを，プログラムのアクセス可能なレジスタに置く．そして，ハードウェアは，スーパーバイザの特定な位置，すなわち，**割り込み解析部**(interrupt analyzer)とよぶルーチンの先頭に飛ぶ．

割り込み解析部は，割り込みの型を識別し，割り込みを処理する適当な**割り込み応答ルーチン**(interrupt-response routine)へ飛ぶ．これは，たとえば**分岐表**(branch table)を参照して行なう．この表は，割り込み応答ルーチンの仮

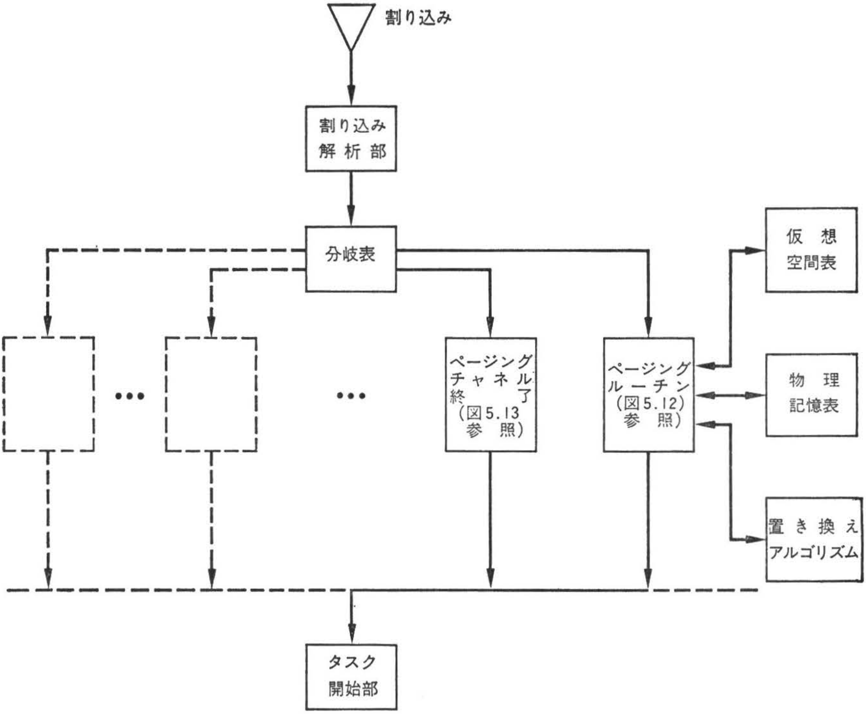


図 5.11　オペレーティング・システムの常駐部

想番地，割り込み応答ルーチンの識別番号をもち，割り込みの識別番号をインデックスとしている(図5.11参照).

割り込み応答ルーチンの番地への飛び越しは，ロケーションカウンタ† を分岐表の適当な場所に単に設定するだけで，ハードウェアの割り込み機構によって直接行なわれる.

プログラムがどのブロックも割り当てられていない仮想番地を参照すると，**ページフォールト**(page fault)割り込みが起る．ページフォールトに対する応答は，**ページフォールト割り込み応答ルーチン** (page-fault-interrupt-response routine)または**ページングルーチン**(paging routine)によって行なう．ページングルーチンは，補助記憶の中に，必要なページの二次番地を見つけなければならない．これを行なうために，処理装置を使っているタスクのすべてのページに対して仮想番地と二次記憶番地との写像を与える**仮想空間表** (virtual space map)をもっていなければならない.

ページングルーチン(図5.12)は，また，必要なページを置くために，物理記憶内で使用できるブロックを見つけなければならない．それゆえ，すべてのブロックの使用の記録を，**物理記憶表**(physical memory map)の中にもっていなければならない．物理記憶の中に使えるブロックが全然ないか，あるいは使えるブロックが非常に少ないときには，ページングルーチンは，**置き換えアルゴリズム**(replacement algorithm)を開始しなければならない．このアルゴリズムは，あるタスクのページを含んでいるブロックを解放する．したがって，ブロックの解放は，タスクのページとそのページが存在しているブロックとの対応関係を破棄する論理的な操作である．もしもタスクがページの内容をブロックに居る間に変更していないのであれば，この操作では，タスクに付随しているページ表内のそのページに対するエントリからそのブロック番号を取り除くだけでよい．タスクがその内容を変更したのであれば，そのブロックが再割り当てされる前に，その内容を補助記憶に退避しなければならない．それゆえ，ページ出力という物理的操作も行なわなければならない.

ページングチャネル終了割り込み応答ルーチン (paging-channel-complete-interrupt-response routine) (図5.13)は，記憶機器が使用中の間に補助記憶の待ち行列に置かれていたページング操作の起動をかける．チャネルの終了は要求ページングの終了であることに注意し，タスクが処理装置で実行可能になったことを指示する．ページングチャネル終了割り込み応答ルーチンはチャネルの終了信号を読み取らなければならないので，これは常駐型になる．実行可

---

† **訳注**　プログラムカウンタ

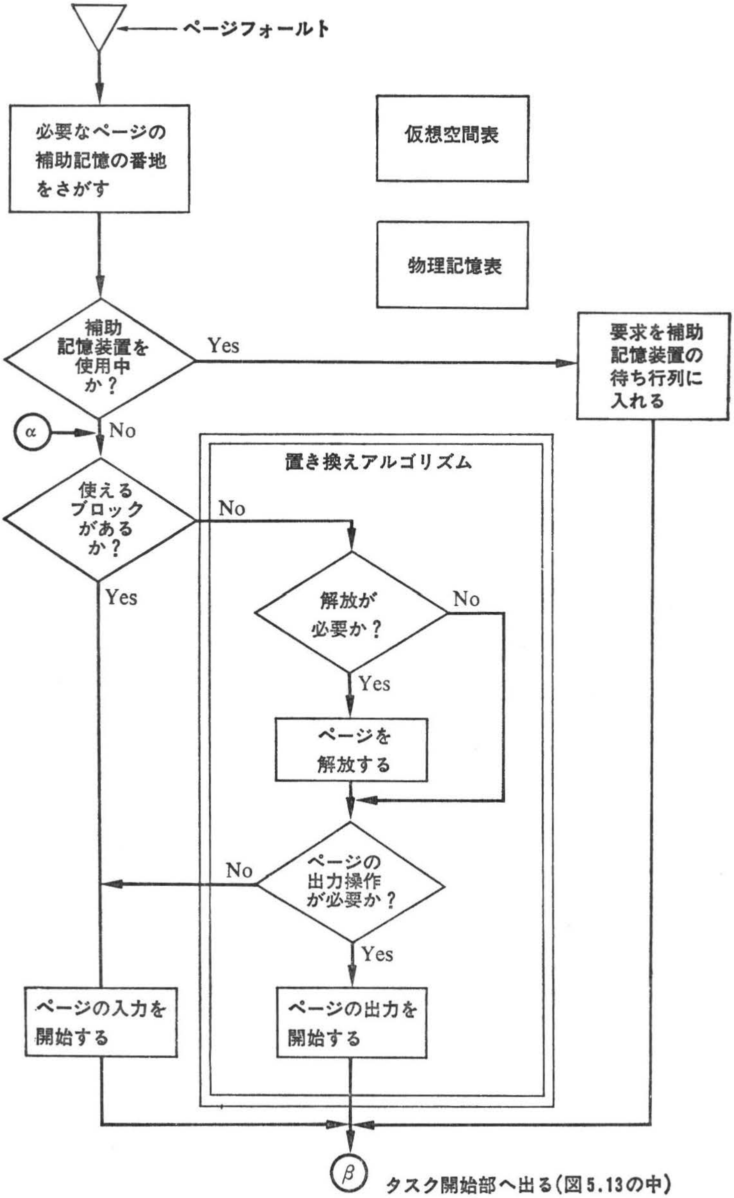


図 5.12　ページングルーチン

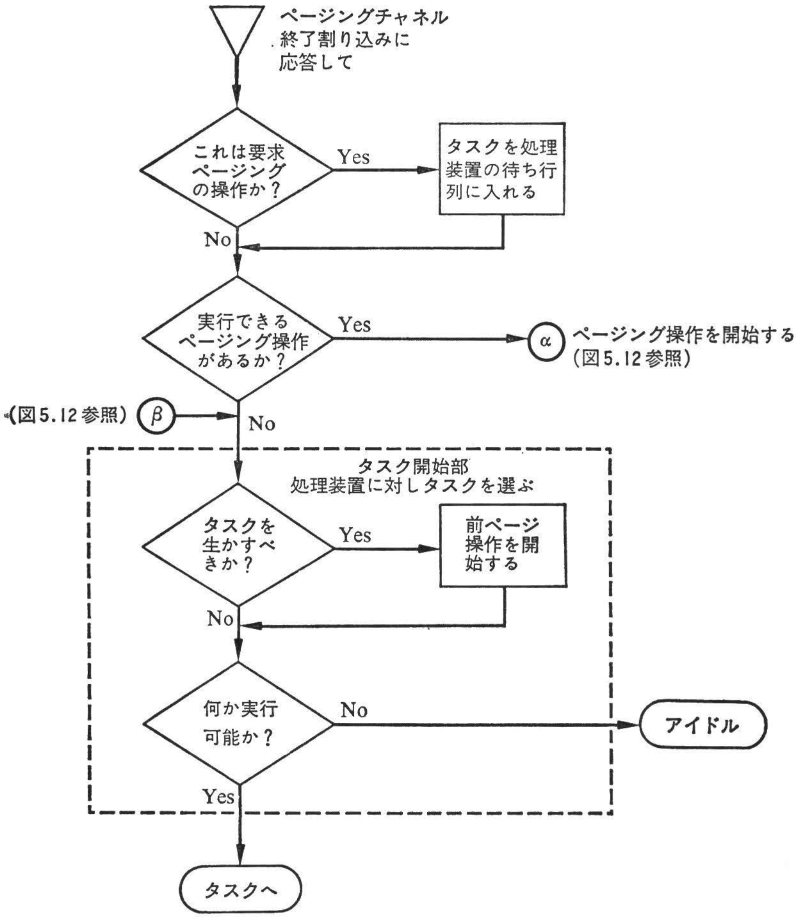


図 5.13　ページングチャネル終了割り込み応答ルーチン

能なタスクを実行するためにすぐに処理装置に戻しても最も能率の良いスケジュールとはならないかもしれないので，いつでもすぐに戻るわけではない．その代りに，このルーチンはタスクを待ち行列に置き，**タスク開始部**(task initiator)とよぶサブルーチンへ飛ぶ．タスク開始部の機能は，処理装置に対する次のタスクを選択することである．このために，タスク開始部は，ディスパッチャまたはスケジューラとも呼んでいる．

タスク開始部は，2つの異なった環境で呼び出される．第一に，タスクは処

理装置を要求するかもしれない．たとえば，ページングチャネルの終了により，現在ページングされているタスクが処理装置を要求していることを指示する．第二に，オペレーティング・システムが，タスクを処理装置に与えるときに**インターバルタイマ**（interval timer）をある時間刻みに設定することにより，タスク開始部をいろいろのスケジュールした間隔で呼び出すかもしれない．指定した時間の後で，インターバルタイマは期限が来て割り込みを起こす．この割り込みに対する応答は，タスク開始部への呼び出しとなり，それがタスクを処理装置で継続させるか，あるいは他のタスクと置きかえるかどうかを決定する．タスク開始部は，また，タスクに割り当てる処理時間の量も決める．その決定は，処理装置の待ち行列におけるタスクの位置，外部優先度，実行時間，割り当てなどに基づいて行なう．この決定の集合は，システムに対するスケジューリングアルゴリズムの主要部分を構成する．

そこで，次のものがその機能によりオペレーティング・システムの常駐部分になるべきものである．

a） 割り込み解析部，
b） ページングルーチンとその表（物理記憶表と仮想空間表），
c） 置き換えアルゴリズム，
d） ページングチャネル終了割り込み応答ルーチン，
e） タスク開始部．

オペレーティング・システムの残りの部分は，仮想空間内に存在でき，要求ページングの下で実行できる．特に，割り込み応答ルーチンは，仮想空間内に置くことができる．物理記憶内に応答ルーチンが常駐していないような割り込みが生じたときには，分岐表内の番地へ飛ぶと，ページフォールトにより別の割り込みを起こす．元の割り込みの処理ができるために，この割り込みとページング操作を完了しておかねばならない．この方法では，すぐに実行すべき機能に対するページングの遅れが頻繁に生ずるので，システムは非能率的になるであろう．

このオペレーティング・システムでは，ページングチャネル終了信号，ページフォールト，インターバルタイマの期限切れによって起るすべての割り込みは，常駐ルーチンで直ちに処理されるであろう．一方，プログラムがひき起こす割り込み，あるいはSVCは，それが呼び出される頻度に応じて，常駐ルーチンで処理されたりされなかったりする．

5.2.2.2 物理入出力操作と PIOCS

A. チャネル待ち行列の管理

入出力機器は，チャネルを介して発せられるオーダによって操作される．一般に，入出力機器や入出力機器の集団は，その機器制御装置を介して，いくつかのチャネルのうちの一つを通じて CPU や物理記憶と通信を行なう．このような柔軟性のために，少ししか使っていないチャネルに操作を行なわせて一つのチャネルだけが混雑するようなことを避けることができるので，入出力機器の操作の効率を非常に増大することができる．一つの入出力機器へのオーダは，そのオーダを受け取れるチャネルがすべて使用中のときにだけ遅れてしまう．入出力機器の主なオーダは，次の通りである．

シーク シークは，物理レコードたとえばシリンダを読んだりサーチしたりできるように，入出力機器のアクセス機構の位置づけをする．位置づけの間，チャネルは空いているか，あるいは他のどれかの機器に対して入出力操作を実行している．

サーチ サーチは，シークの完了したあと，特定の物理レコード(たとえば，トラック)を捜す．サーチでは，特定のレコードの位置を決めるために，物理トラック上のいくつかのデータ(キーフィールド)を取り出すことになる．チャネルは，サーチの間は使用中となる．

読み/書き 適切な位置が見つかれば，データを（トラック）領域から読み込んだり，(トラック)領域へ書き出したりする．

多重パスの配置をして特に利益のあるのは，入出力の使用が激しいときにシークオーダを実行できることである．シーク操作は，チャネルをずっと使っている必要はない．機器制御装置にオーダを送るために，一瞬だけチャネルを使えるだけで良い．したがって，要求された入出力機器が使用中でなければ，その機器に対するシーク操作は，その機器と通信できるチャネルのどれか一つが操作を完了するまで遅れるだけで済む．この遅れは，一般に，唯一つだけのチャネルをその機器に接続している場合に比べてずっと小さい．

入出力操作は，次の2つの理由のうちのどちらかによって，待ち行列に入れられる．第一に，その入出力機器がシークやサーチの操作を実行しているところかもしれない．この場合，この操作は，その機器の待ち行列に置かれる．第二に，その機器に対するすべての使用可能なパスがすべて使用中かもしれない．そこで，この操作は，その機器に対するパスの集合に対する待ち行列に置かれる．入出力機器の**パスの集合**(path set)とは，その入出力機器と通信でき

るすべてのチャネルの集合と定義する．逆に，ある与えられたチャネルのパスの集合に対し，これらのチャネルに接続している機器を，パスの集合に付加しているという．したがって，一つのチャネルは，そのチャネルに接続されている各機器のパスの集合に属しており，いくつかのパスの集合の要素となる．パスの集合は，その集合の中のチャネルがすべて使用中であれば，使うことができない．少くともその中の一つのチャネルが使用されていないときに，使うことができる．パスの集合の例を，図 5.14 に示す．

パスの集合の各々に対して，2つの待ち行列がある．一つは，そのパスの集合に付いている使用可能な機器に対するシーク操作の待ち行列である．このパスの集合に付いている同一の機器に向けられた2つの異なったシーク操作がこのパスの集合の待ち行列にあらわれない限り，この待ち行列は，パスの集合の一つのチャネルが使用可能になったときに，空になる．この場合，すべてのシークのオーダを，このチャネルを介して送ることができる．もう一つは，サーチ操作の待ち行列である．このパスの集合の一つのチャネルが使用可能になったとき，待ち行列内の一つのサーチ操作を選択する．その選択の基準は，タスクの優先度，パスの集合の優先度，および，アクセス機構の位置，回転遅れの周期，入出力操作の予想される長さなどのような入出力機器と操作の要求に基

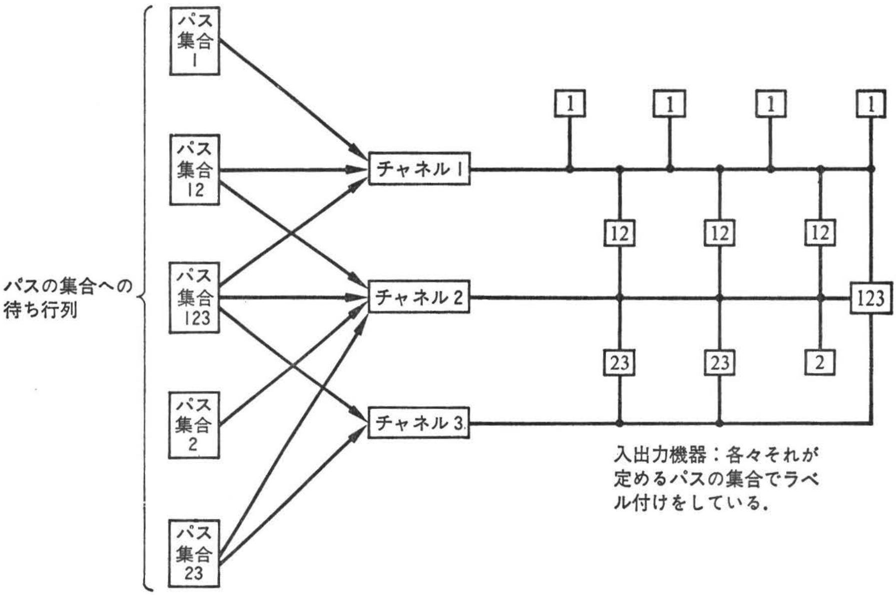


図 5.14 パスの集合の例

づいて決まる．一般に，小さなパスの集合の処理の方が機会が少ないと思われるので，このアルゴリズムでは，少ないチャネルのパスの集合の方を多いものよりも優先する．この選択の優先度は，究極的には各パスの集合に属する機器の数とこれらの機器に対する参照の頻度に依存するであろう．

チャネルの操作は，PI/O マクロを呼び出したときに始まる(次の節の PI/O マクロを参照せよ)．入出力操作が要求されたときその機器あるいは機器のパスの集合が使用中であれば，その操作を対応する待ち行列に置く．これらの操作は，一つの操作が終了して機器やチャネルが空いたときに，開始しなければならない．

B.　入出力機器の操作

多重プログラミングの環境の下では，ファイルに使う二次記憶装置を共用している．ファイルの保全性を守りより効率の良い共用を実現するために，物理入出力命令を特権化し，それを操作系から出させる．ユーザのタスクで入出力操作が必要になったら，**物理入出力 SVC**(physical-I/O SVC)とよぶ SVC 命令を使う．これによって割り込みが起り，簡略化して **PI/O マクロ**(PI/O macro)とよぶ**物理入出力応答ルーチン** (physical-I/O-response routine)がそれに応える．その機器が使用できれば，PI/O マクロは，チャネルコマンドと機器オーダを出して，機器の操作を開始する(図 5.15)．機器が使用できなければ，マクロにより操作を待ち行列に置いて，後で機器が使用可能になったときに実行する．

PI/O マクロに対する入力は，第2章で詳しく述べた機器制御ブロックに入れておく．ここでは，これについて議論を繰り返すことはしない．しかし，ここでは，PI/O マクロの操作を**資源割り当て**(resource allocation)の見地から論じてみよう．

3つの形のオーダ——すなわち，シーク，サーチと読み書き——が典型的な入出力操作で必要になるけれども，処理装置は各オーダをチャネルを介して機器に陽にかつ単独に送り出す必要はない．オーダはチャネルコマンドと一緒ににチャネルプログラム(channel program)にしておき，チャネルは現在の入出力オーダが完了すると新しいオーダをそこから発生させる．物理入出力マクロは，そのようなチャネルプログラムを設定して，チャネルプログラムの実行命令を出すことを行なう．

このオペレーティング・システムでは，ディスクのような直接アクセス機器に対する入出力操作は2つのステップに分割され，それらに対し2つの命令

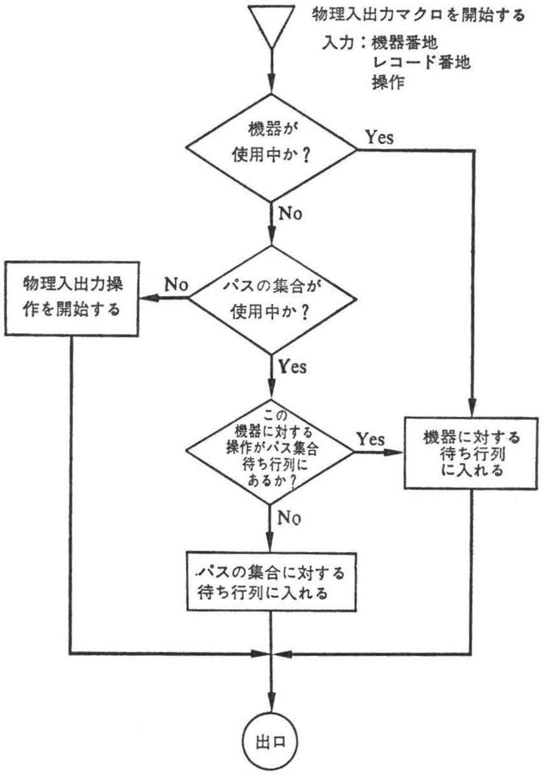


図 5.15 物理入出力の開始

(2つのチャネルプログラム)が出される．第一に，シークのオーダから構成されるチャネルプログラムを出す．位置づけ操作が完了すると，チャネルは，チャネル終了割り込みを起こし，そのチャネルプログラムが終了したことを知らせる．その後，割り込み応答プログラムがサーチや読み書き操作の命令を出す．したがって，物理入出力操作のそれぞれに対し，2つのチャネル終了割り込みが生じ，チャネル終了割り込みの応答ルーチンは，どのチャネルプログラムが終了したかどうかに応じて，異なった動作をとる．

直接アクセス機器に対する入出力操作が2つのステップで実行される理由は，次の2つである．第一に，機器やチャネルの使用の完全な記録を得るために，シーク(これに対してはチャネルは使わない)が終了したり，サーチが始まったときに記録をしておくのは役に立つであろう．第二に，これによって入出力のスケジューリングが，非常に柔軟にできる．たとえば，シークに対するチ

ャネルプログラムを実行しているチャネルで優先度の高い入出力操作を開始しなければならないときには，その次のサーチに対するチャネルプログラムは，優先度の高い入出力操作のシークとサーチの両方が終了するまで遅らせることができる．

このような方法で入出力操作を実行するために必要な費用は，各操作に対する余分な割り込みである．大抵の場合，優先度の高い割り込みは存在しない．したがって，シークのプログラムが完了したとき，チャネル終了割り込み応答ルーチンは優先度の高い入出力操作があるかどうかだけを調べて，それから既に準備されている第二のステップに対する命令を出す．それゆえ，余分な割り込みは，処理装置の時間を少ししか必要としない．

C. チャネル終了割り込みの応答

チャネル終了割り込みは，シークあるいはサーチと読み書き操作のどれかが終了したときに起る．シークの終了によって起る割り込みは，CPU へ割り込みを送るためにチャネルが使えるということを意味する．使用可能なチャネルではシーク操作を遅らせる必要はないので，シーク操作がそのチャネルの待ち行列に残っていないことも意味している．したがって，シーク操作の終了を示す割り込みの後すぐにサーチや読み書き操作を始めることができる．

もしも割り込みが読み込みや書き出し操作の終了によるものであれば，チャネルを使用中にほかの操作がその待ち行列に入ってきたかもしれない．第一に，他のシーク操作が終了しているかもしれない．これらの操作に対して(禁止されていた)割り込みが，チャネルが空くまで機器や機器制御装置で保持されている．**チャネル終了割り込み応答ルーチン** (channel-complete-interrupt-response routine) (図 5.16) はこれらの禁止されていた割り込みを見つけ，この割り込みを処理し対応するサーチ操作を空いたチャネルを要素として含んでいるパスの集合の待ち行列に入れる．

第二に，ちょうど終了した機器に対する待ち行列に，別の操作があるかもしれない．一般に，それは新しいシークであろう．しかし，もしもそれがちょうど終了した操作と同じシリンダに対するものであるならば，サーチの部分だけ必要となる．それがシークであれば，すぐに始める．もしサーチならば，他のサーチ操作と一緒にしてパスの集合に対する待ち行列に入れる．

第三に，チャネルを使用中に新しい入出力操作が要求されていたかもしれない．この時点までパスの集合が全部使用中であったような新しい要求は，各々そのパスの集合に対する待ち行列の中でシーク操作として表わされている．そ

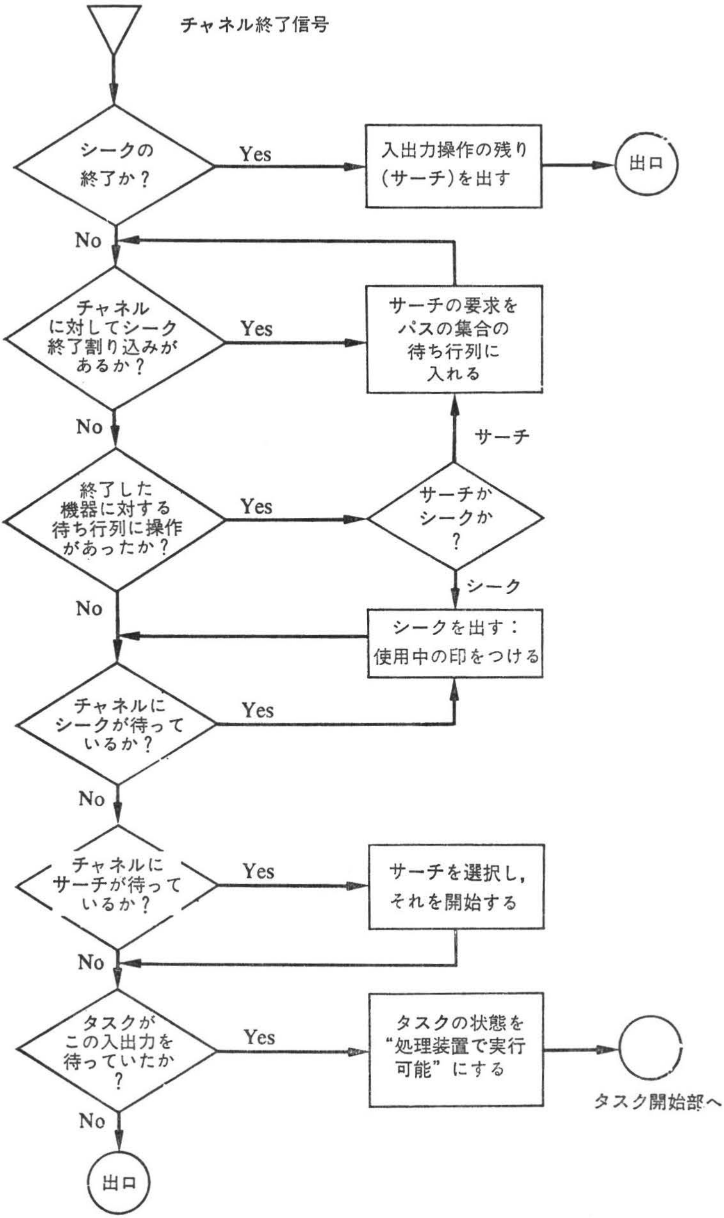


図 5.16 入出力チャネル終了割り込みの応答

のパスの集合の待ち行列で一つの機器に対して多数の操作がある場合は，一つの操作を選択して開始し他の操作は機器に対する待ち行列に入れておく．

最後に，このチャネルに対する待ち行列の中のサーチ操作から，一つのサーチ操作を選ぶ．この選択は，前に述べたように入出力のスケジューリングを考慮して決める．

D.　タスク開始部

タスク開始部は，どのタスクが処理装置の制御を得るのかを決めるシステムプログラムである．それは，実行可能なタスクのプールからタスクを選択する．タスクは，いくつかの理由により処理装置で処理できないかもしれない．

1.　要求したページが物理記憶の中にはないので，タスクはページングの待ち行列に入っている．
2.　タスクは，入出力操作で要求したデータを待っているところである．この期間は，このタスクに与えられたブロックは，勿論，置き換えアルゴリズムに従って別のタスクに取られるかもしれない．
3.　タスクは，端末からの応答を待っているところである．この待ちは，長い時間にわたると予想されるので，このタスクに対してシステム資源は何も割り当てていない．このタスクを，**眠っているタスク**(dormant task)という．

1. の場合，ページフォールトの割り込みの知らせにより，システムはタスクの処理を中断する．そして，システムのページフォールト応答ルーチンは，処理装置で実行できるという状態からタスクを除く．

2. の場合，生きているタスクが入出力操作の終了を待たなければならないときには，“アイドル”のループには入らない．その代り，タスクは，**入出力待ち**(I/O wait)SVC を出す．これは，このタスクを処理装置で実行できる状態から除き，タスク開始部は別のタスクを CPU に与えて良いことをシステムに伝える．

同様にして，**3.** のようにタスクが端末からの応答を待たなければならないとき，**端末待ち**(terminal wait)SVC を出す．端末待ち SVC に対する応答として，オペレーティング・システムはそのタスク（恐らくそのタスクのページは掃き出されるかもしれない）を眠りの状態にして，タスク開始部に移り処理装置に与える別のタスクを捜す．

ページングや入出力の待ちのために処理を止められてしまったタスクは，ペ

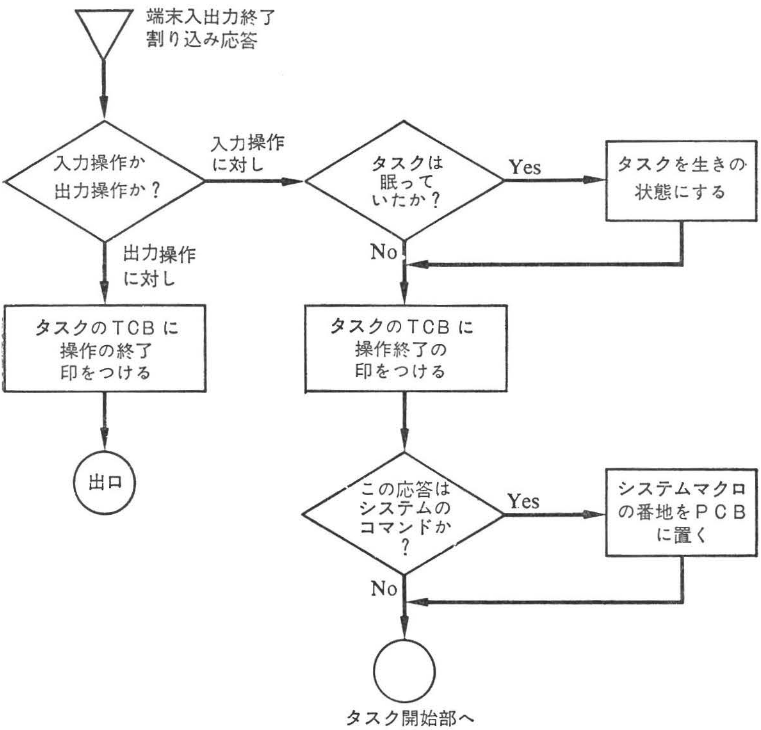


図 5.17　端末入出力終了割り込みの応答

--ジングチャネルや(入出力)チャネル終了割り込み応答ルーチンによって，生きの状態に戻される．眠っているタスクは，端末からの入力メッセージの終了を示す割り込みに応答するルーチンによって，生きの状態に戻る．このルーチンを**端末入出力終了割り込み応答** (terminal-I/O-complete-interrupt-responce)ルーチンとよぶが，この概略を図 5.17 に示す．

要約すると，物理入出力操作は，準備するのがたいくつで管理するのが面倒なものであることを指摘してきた．それにも拘わらず，このような準備や管理はオペレーティング・システムの機能である．割り込み応答ルーチン，チャネルや機器の管理ルーチン，機器制御ブロック，パスの集合の待ち行列などが，オペレーティング・システムの**物理入出力制御サブシステム** (physical-I/O control subsystem: **PIOCS**)を構成する基本要素である．

5.2.2.3 論理入出力操作，LIOCS と ファイルシステム†

多重プログラミングシステムでは，ユーザプログラムはシステムのハードウェアに対して直接制御をすることはできない．そこでは，二次記憶機器の物理番地や機器上のデータ領域の番地を使って入出力操作を指定することはしない．その代り，オペレーティング・システムは，ユーザのデータに対し記憶機器の場所を割り当て，ユーザに対しこれらのハードウェアの番地を必要とするようなハードウェアのコマンドを出す．さらに，オペレーティング・システムは，データに対するユーザ名とデータに対する物理的場所との対応を確立しその管理をする．データのユーザ名と物理的場所との対応は，**論理入出力** (logical I/O: **LI/O**) 操作とよぶシステムマクロの集合によって行なわれる．（第2章の論理入出力に関する演習問題を参照せよ）．もっと詳しくいうと，ユーザのデータは，論理レコードとして扱う．各論理レコードには，ユーザプログラムが英数字の**キーワード** (keyword) を割り当てる．レコードを二次記憶に格納するために，ユーザは最初にレコードに対する**ファイル** (file) を作成し，そのファイルに**ファイル名** (file name) を割り当てなければならない．ファイルの生成により，オペレーティング・システムは入ってくるレコードに対する二次記

---

† 論理入出力操作は，多重プログラミングシステムだけでなく一括処理システムでも必要である．ファイルシステムは，第3章にも示したように，オンライン一括処理システムと多重プログラミングシステムにおける必要な部分である．ファイルシステムと論理操作の議論をこの章まで遅らせてきたのは，次の理由のためである．

1. 一括処理システムと多重プログラミングシステムで仮想空間を持ったものとそうでないもの両方における論理入出力操作は，伝統的に似ているものである．論理入出力操作をより一般的な環境（すなわち，仮想記憶システムの環境）の下で示すことにより，一括処理および多重プログラミングシステムにおける論理入出力操作の使い方を特殊化でき導くことができる．したがって，前の章で述べた題材をここで繰り返す必要はない．
2. 仮想記憶システムにおける論理入出力操作を示すには，仮想空間を理解しなければならない．前の章で論理入出力操作を示そうとしても，仮想空間の概念や術語が導入されていなかったので，どうしても不完全なものになってしまったあでろう．
3. 伝統的な論理入出力操作は仮想記憶システムに支援されていても，それは，主として，プログラムの互換性の必要からでてくるものである．すなわち，仮想空間のない以前のシステムで開発した論理入出力を使う古い（ユーザの）プログラムが，仮想空間をもった新しいシステムで実行できなければならないので，それらを仮想記憶システムの上に実現してきた．たとえば，IBM 370 OS/VS 2-1 は，IBM 360 MFT で完成された論理入出力操作を支援している．同様に，IBM 370 OS/VS 2-2 は，IBM 360 MVT の論理入出力操作を支援している．これらの場合，IBM 360 上のユーザプログラムは，論理入出力操作の非互換性を気にしなくても，IBM 370 上へ移すことができる．しかし，もっと効率の良い操作もある．その1つは，**仮想アクセス法** (virtual access method) とよばれ，仮想記憶システムの上だけで使用できる．したがって，VAM の議論に沿って論理入出力操作を議論すると，より効率の良い入出力を照らし出してくれるかもしれない．新しいシステムのユーザは，新しいプログラムにおいて VAM を使い，除々にその古いプログラムの中の論理入出力を VAM に置き換えてゆく．

ファイルシステムの議論をこの章で行なう理由づけも，論理入出力に対して行なったものと同様である．

憶を割り当てる．ファイル名を割り当てると，ユーザは，レコードの集合を論理的な実体として参照することができる．したがって，ユーザプログラムでは，二次記憶の番地を知る必要はなくファイル名とキーワードを用いてレコードを参照することにより，論理入出力 (LI/O) 操作を使ってレコードの格納や検索ができる．ファイル名，レコードのキーワード，レコードおよび番地の間の対応関係は，ファイルシステムが維持するが，これは次の節で論じよう．レコードの格納や検索をし，**論理入出力制御サブシステム**(logical-I/O control subsystem: **LIOCS**)の主要な部分を構成する論理入出力操作を実現することは，できるであろう．

A. 単純なファイル構造

二次記憶の中のファイルは，3つの部分，すなわち，定義，ディレクトリおよびレコードから成り立つ．**定義**(definition)は，ファイルの生成，アクセス制御の要求，ファイルディレクトリの二次記憶の番地に関する情報をもっている．ファイルの**ディレクトリ** (directory) は(キーワード，物理番地)の対の集合で，ファイル内の各レコードのキーワードから二次記憶内のレコードの物理番地への写像を定義している．単一のトラックに含むのに大きすぎる場合は，多重レベルの構造になるであろう．(わかり易くするために，物理レコードとかブロックという術語は使わない．前者は，論理レコードと混同する．後者は，ページングにおける主記憶のブロックと混同する)．ファイル内のレコードのキーワードが英数字の順に並んでいて，(キーワード，物理番地)の対の集合が1トラックを越えてしまうならば，図 5.18 に示すような2レベルのディ

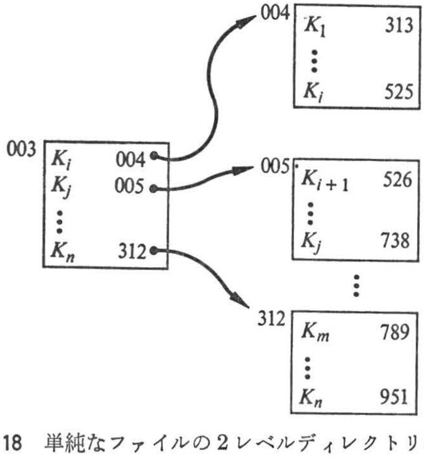


図 5.18 単純なファイルの2レベルディレクトリの実現

レクトリが必要となる．第1のレベルは，(高キーワード，第2レベルのトラック番地)の対で，高キーワードは対応する第2レベルのトラックにおける最大(かつ最後)のキーワードである．第2のレベルは，写像を定義する対で，必要なだけ多くのトラックに拡大してゆく．

K_i がキーワード i をあらわすとすると，次のようになる．

ファイルのレコードは，トラック上に格納する．2つ以上のレコードが単一のトラックに格納されたときは，トラック上のレコードをキーワードによって辞書式順序に並べる．ファイルの構造は，次の方法により再帰的に拡張することができる．一つのファイルのレコードは，別のファイルの定義になってもよい．図5.19を見ると，Bのファイル定義は，ファイルAのレコードであることに気が付く．したがって，ファイルAはファイルBの**プレデセサ**(predecessor)であり，BはAの**サクセサ**(successor)であるという．記号的には，A＝P(B)と書く．ユーザプログラムは，ファイルBを**修飾ファイル名**(qualified file name)A.Bによって参照してよい．

B.　ユーザファイルの番地変換

システムカタログ (system catalog: **Syscat**) とよぶすべてのファイルの一覧表を，システムが管理している．これは，均質でない木の構造をしている．すなわち，Syscat の根 (定義が他のファイルのレコードでないファイル) から

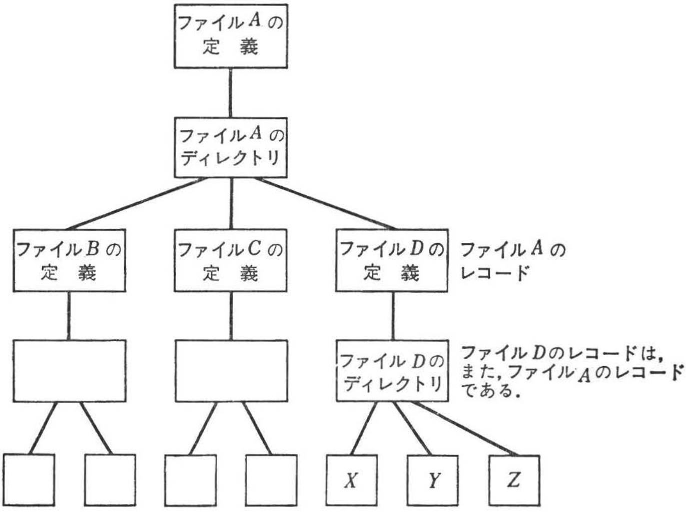


図 5.19　ファイルAの構造

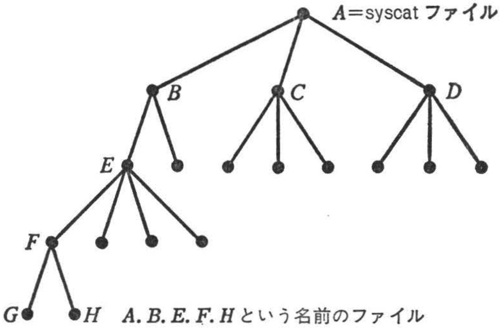


図 5.20 名前づけの方法の抽象的構造

始まって，終端(ユーザファイル)にまで到達するために捜すべきファイルの数は，各終端ごとに同一ではない．図5.20に，Syscat の表現を木として示す．

特定のファイルは完全な修飾ファイル名で指定する．すなわち，その名前は，いくつかのファイル名——それぞれは，そのファイルを捜すために Syscat 木の中でとられる分岐を指定する——の連接から構成される．たとえば，図5.20 で，ファイル名 A.B.E.F.H は完全な修飾名である．これは，一つのユーザファイルを指定する．名前 A.C は部分的に修飾している．これは，ユーザファイルのファイルを指定する．

特定のファイルを所有しているユーザの名前が完全な修飾ファイル名のある部分にあらわれる．したがって，あるファイルの所有者と同じファイル名が，Syscat のどこかのレベルにあらわれる．このファイルは，Syscat 木の部分木の根であり，これは，そのユーザが所有しているすべてのファイルが要素となっているファイルを定義している．この根は，ユーザの**ファイルのファイル**(file of files: **FOF**)とよぶであろう．

Syscat の第一レベルは，多重レベルのディレクトリを持つファイルである．ディレクトリの第一のレベルは，仮想空間に常駐し，残りのディレクトリのレベルは，二次記憶機器に常駐している．Syscat のレコードは，ユーザの個人用および共用ファイルのファイル定義である．

図5.21は，典型的なシステムカタログを示している．ここでは，ユーザの所有するファイルは，各々2レベルであり，公開ファイルは1レベルである．したがって，Syscat の第一レベルのレコードは，多数のユーザの FOF の定義か，または，公開ファイルの定義かのどちらかである．

これが，論理入出力操作の働きを示すために使う Syscat の構造である．これは，多重レベルの構造という特徴をもつ．しかし，ここでの議論のために

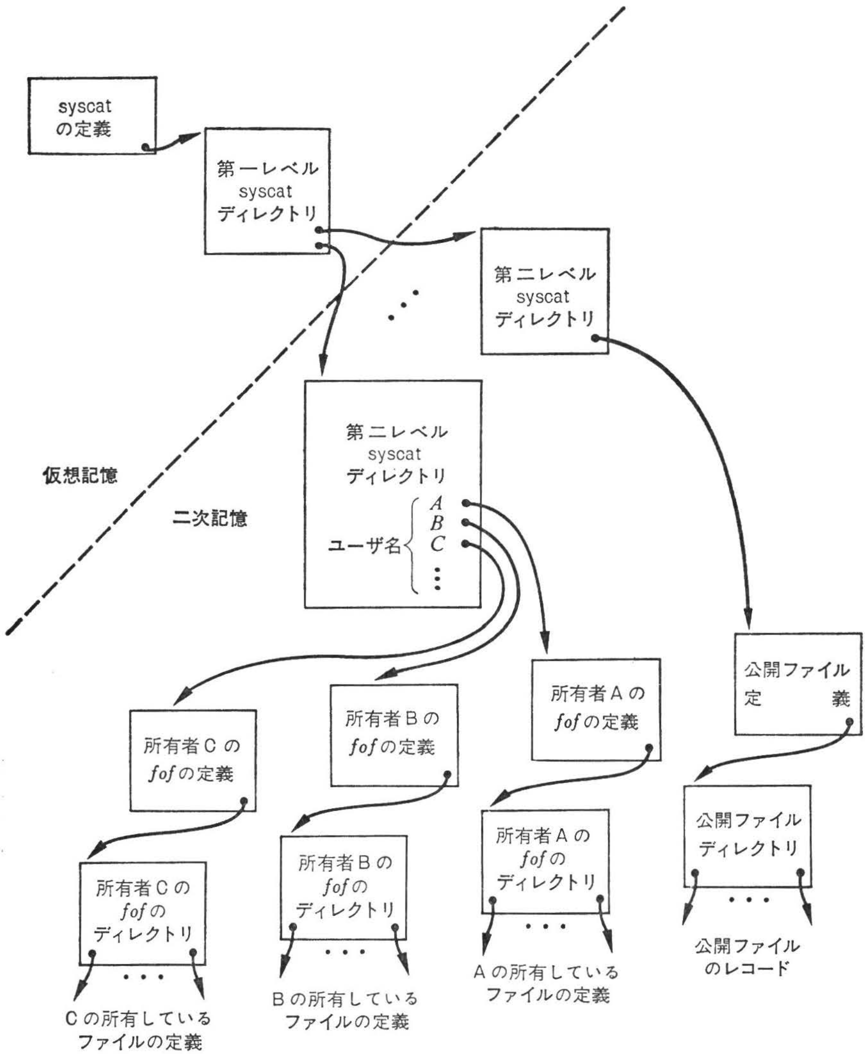


図 5.21 Syscat の構造

は，2レベルのファイル構造で十分である．任意の多重レベル構造に一般化するのは，すぐに可能である．

C. 論理入出力操作

一般に，ユーザのタスクは，GET および PUT マクロに対する SVC を出し

て，論理入出力操作を要求する．これらのマクロは，それぞれ，ファイルからレコードを取り出したり，あるいはそこへレコードを格納する．

各マクロに対する入力は，(1) レコードが移される仮想番地，(2) ファイル名，および，(3) レコードのキーワードである．

ファイルが入出力操作のために生きている間，システムはファイルの状態に関係する情報を**ファイル制御ブロック** (file control block: **FCB**) へ保持している．FCBは，ファイル名，ファイルディレクトリへのポインタ，および，事前の入出力動作の結果として仮想空間中に見つけることができるファイルの部分に関する状態情報を含んでいる．後者は，論理入出力マクロによって参照され，そのマクロに対して実行しなければならない物理入出力操作を決める．

GET および PUT マクロは，操作すべきファイルが事前にオープンされていることを仮定している．そうでなければ，誤りの状態となる．基本的には，ファイルのオープンの操作は，ファイル F のディレクトリの番地を FCB に置くという効果をもたらす．マクロ

GET　F, X

は，次のステップを実行する．

a) ファイル F のディレクトリを仮想空間内にもってくる．
b) キーワード X であるレコードを含むトラックを，ディレクトリ内で見つける．
c) このトラックを仮想空間にもってくる†．
d) レコード X を，仮想空間中で再配置する．(簡単に言うために，レコード X とは，キーワードが X であるようなレコードを意味する)．

次に GET マクロがたとえば GET F, Y のように呼ばれた時には，ディレクトリ F が既に仮想空間に存在しているので，ステップ(a)を実行する必要はない．同様に，レコード Y がレコード X と同じトラックにあれば，そのトラックは仮想空間内に存在しているので，ステップ(b)を実行する必要はない．

ファイル中にレコードがないときには，マクロの出力パラメータにそれが設定される．GET マクロの概略を図 5.22 に示す．

したがって，GET マクロは，ファイルディレクトリやレコードのトラック

† ハードウェアが必要なレコードをさがしてそれだけを仮想空間にもってきてもよいので，全トラックを仮想空間内にもってくることを強制しているわけではない．しかし，この方法は，ファイルの順次処理に対する最大の効率をあげる方法と合致する．順次処理では，レコード X を得ると，同じトラック上にあるレコード Y や Z が直ぐに必要になり，したがって，全トラックを読むことにより，レコード X と一緒にもってきてもよい．

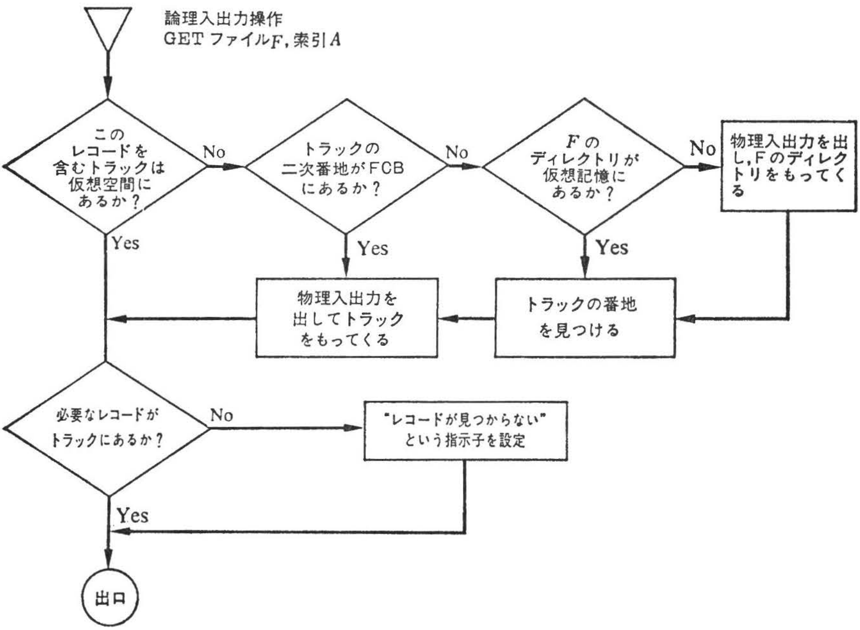


図 5.22　論理入出力の GET 操作

を仮想空間にもってくるために物理入出力命令を出す前に，ファイルのどの部分が仮想空間中に存在しているかを決定する．

ファイルはそれに対する GET マクロが呼ばれる前にオープンされていなければならないので，ユーザタスクは，SVC の伴った OPEN マクロを出してその動作のためにファイルを準備する．OPEN マクロの概略を図 5.23 に示す．

OPEN マクロは，ファイル F の定義を得るために，GET マクロを呼ぶ．この呼び出しに対する入力パラメータは，ファイル名とレコードインデックスで，次の形式 GET P(F), F をとる．(P(F)は，F のプレデセサを意味することを思い出そう.) そこで，OPEN マクロは，関連のあるデータをファイル定義から FCB へもってきて，ファイル F はオープンされる．

もしも，FOF が OPEN マクロの呼び出しの入力であったならば，ファイル P(FOF) は実は Syscat である．ファイル F が P(F) の中に見つからなければ，ファイル名 F は修飾していないファイル名と考えて，ファイル F の探索を Syscat(第 2 レベル)に対して始める．

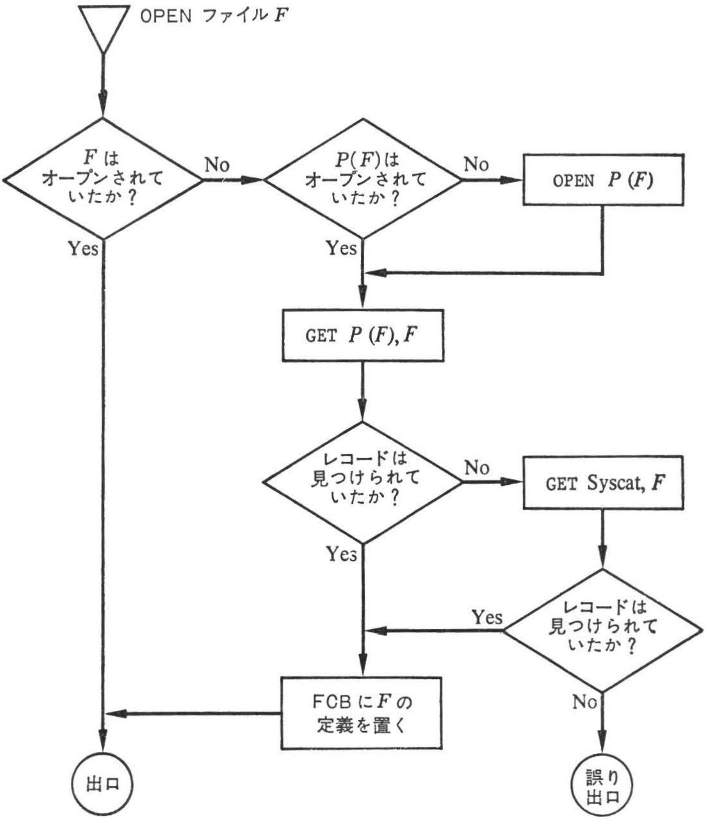
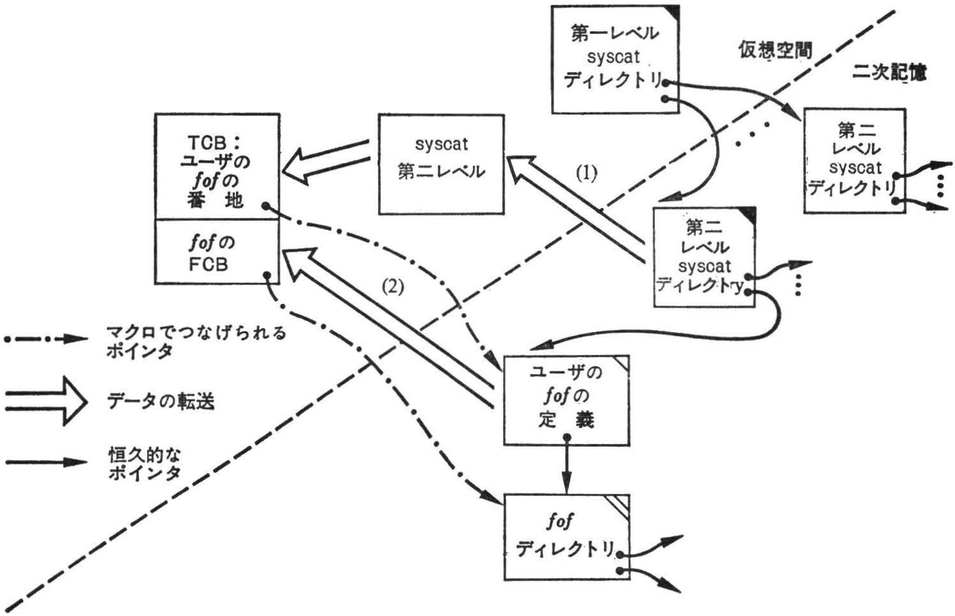


図 5.23 OPEN マクロ

OPEN および GET マクロの全体の操作の概略を，**図 5.24** の例で示す．ここでは，タスクの定義されているユーザと，タスクが必要としているファイルの所有者とは同一であると仮定する†．ユーザが最初にファイルをオープンするとき，そのユーザの FOF をオープンする．GET マクロでユーザの名前を指定して，Syscat を探索する．**図 5.24**(a)の(1)の操作で，Syscat の第二レベルを仮想空間にもってくる．そして，(2)の操作で Syscat レコードをもってくるが，これは，ユーザの FOF の定義である．

GET P(A), A マクロにより，操作(3)で FOF のディレクトリ，すなわち，

† これは必ずしも正しくない．しかし，2つの個人の名前を LOGON の処理で同一視してもよく，所有者の名前を，TCB の中で使うことができる．この操作は，ここではふれない保護とプライバシーの手続きによってきまる．



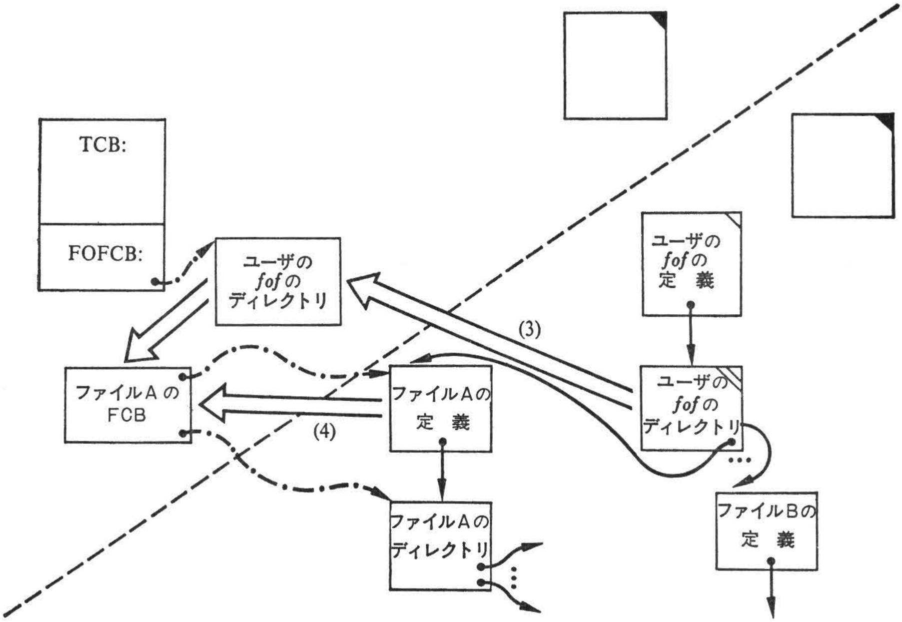
(a) "GET Syscat, user name" が "OPEN fof" から呼ばれ，
それが今度は "GET P(A), A" から呼ばれる．

図 5.24(a)　ファイルの OPEN と GET における物理的操作

すべてのユーザのファイルの名前と場所の一覧表を持ってくる(図 **5.24 (b)** 参照)．この一覧表は仮想空間内に常駐しており，**ユーザカタログ**(user catalog: **Usercat**)として使う．これ以後，ユーザが指定するすべてのファイル操作は，Syscat の中でファイル名を見つける前に，先ず Usercat を参照する．

ファイル A の二次番地を Usercat に見つけた以後，ファイル A のファイル定義は，操作(4)により仮想空間内にもってこられる．それから，ファイル A をオープンする．次に，Usercat を変更して，ファイル定義の二次番地を FCB 内のファイル A の仮想番地と置き換える．

最後に，マクロ GET A, X は図 **5.24(c)**に示す2つの操作につながる．操作(5)は，今度は，Usercat の中でファイル A のディレクトリの番地を見つけ，そのディレクトリを仮想空間にもってくる．操作(6)では，X でインデックスを与えられるレコードを含んだトラックを，仮想空間中にもってくる．その後，ファイル A のレコードに対して GET マクロが再度呼ばれても，GET マクロで使われるすべての物理入出力操作が必ずしも必要となるわけではない．

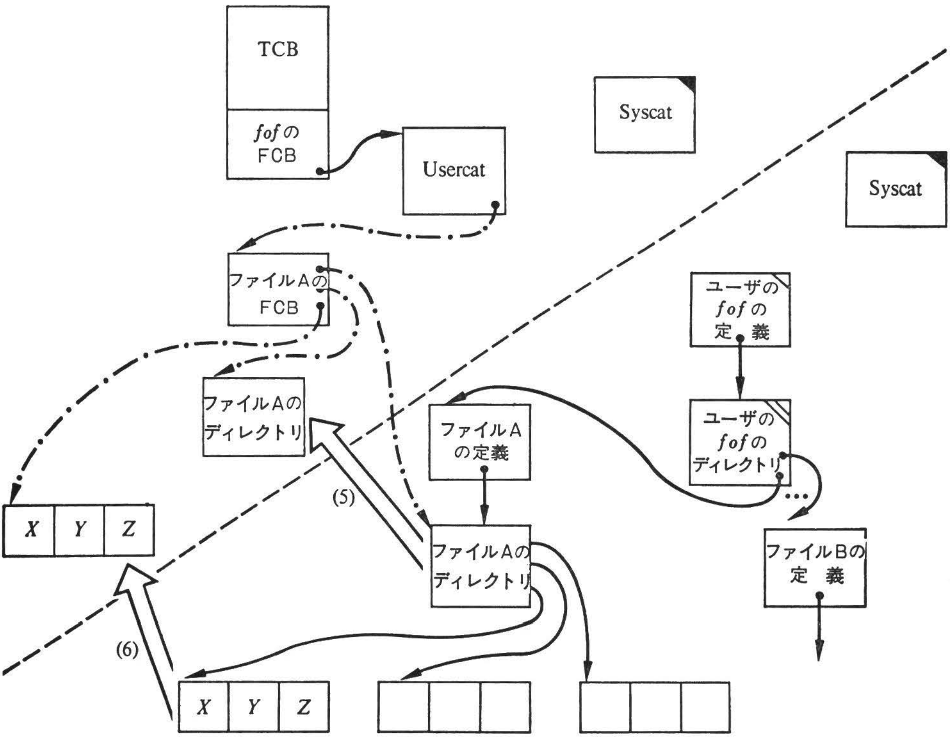


(b) OPEN A が終了した後

図 5.24(b)

PUT マクロは，レコードをファイルへ格納する．その操作は，GET マクロと非常に似ている．ここでは，それを論ずることはしない．

要約すると，論理入出力操作は，普通物理レコード(たとえばトラック)の大きさよりも小さいレコードをもつファイルに対する操作であることに注意しよう．大きな物理レコードを検索しデブロッキングをすることにより，論理レコードを GET 操作によりユーザのタスクに分配することができる．一方，小さい論理レコードを PUT 操作により物理レコードになるようにブロッキングする．一方で入出力チャネルと機器に対する高転送率を利用し，もう一方で二次記憶機器の長いアクセス時間を減少させるためには，大きなブロッキング係数(したがって大きな物理レコード長)を使うように設計する，しかしながら，これらの設計上の考慮も，仮想記憶システムにおける論理入出力操作には何の効果もない．入出力に対するブロックの大きさ(たとえばトラックやシリンダの大きさ)は通常非常に大きいのに，ページングにおけるブロックの大きさ(すなわち，ページの大きさ)は小さいので，この問題は特に厳しい．論理入出力操



(c) GET A,X が終了した後

図 5.24(c)

作に対し大きなブロックを要求し，大きな物理レコードに対するデブロッキングとブロッキングとを仮想空間で行うと，非常に多くのページ(したがって物理記憶ブロック)が必要となる．したがって，伝統的な論理入出力操作を仮想記憶システムに対して改良しようという試みが行なわれてきた．これらの試みについては，次節で詳しく論じよう．

GET, PUT, OPEN およびその他の論理入出力マクロは，論理入出力制御サブシステム(LIOCS)の基本要素を構成する．ブロッキング，デブロッキング，バッファの切り換えおよび管理(たとえば，すべてのバッファのページを物理記憶ブロックに保っておくこと)などを実行する LIOCS 機能は，通常，**入出力バッファリングサブシステム**(IOBS)と見做されている．二次記憶機器は使う前に識別しなければならないので，LIOCS はこれらの機器(たとえば取り

はずし可能なディスクパック）をラベルによって検証しておく必要がある．ラベルを検索し，書き出し，検証する LIOCS の部分を，共通に**入出力検証サブシステム**(IOVS)とよぶ．(IOBS と IOVS の議論については，読者は第2章を参照されるとよい）．

オンライン一括処理システム，多重プログラミングシステムおよび仮想記憶システムでは，オンライン記憶機器と中央の情報貯蔵所を使うことが必要となっている．情報は，ファイルとレコードに組織化されている．ファイルとレコードを参照するのは，名前とキーワードで行なう．ファイルとレコードがどこに居るのかを管理するために，ディレクトリを作成する．特に，すべてのディレクトリ，ファイルおよびレコードの親ディレクトリとなるシステムカタログは，オペレーティング・システムの一部となっている．カタログ，ディレクトリ，ファイル，レコードと，それらを保持する機能を合わせて，**ファイルシステム**(file system)とよぶ．

5.2.2.4 仮想アクセス法

これから述べる**仮想アクセス法**(Virtual Access Method；略して **VAM**)は，仮想空間の構成とシステムの番地変換機構を利用した論理入出力操作の集合である．したがって，これは仮想記憶オペレーティング・システムに特定のものである．あきらかにこの方法で使う仮想空間の構成および変換機構の影響が重要である．ファイルの構造とレコードの構成を再構築しなければならない．便宜上，これらのファイル，ディレクトリ，およびレコードを VAM ファイル，VAM ディレクトリ，および VAM レコードと呼ぼう．それにも拘わらず，前節に述べたカタログの形式は，ここでもやはり適用することができる．

VAM ファイルの VAM レコードは，仮想空間の複数個のページに拡がってもよい．したがって，レコードは可変長である．VAM レコードの1ページは，残りのレコードにある特定の項目を参照する**内容表**(table of contents; **TOC**)を含んでいる．VAM レコードを呼び出すとき，ユーザは検索してくるレコードを置いておく仮想番地を指定しない．その代り，アクセス法(マクロ)が必要なすべての仮想番地を与える．アクセス法は，また，タスクにレコードに対する TOC の番地を与える．TOC は，レコードの先頭からの相対的な番地をもっている．したがって，ユーザタスクは，VAM レコードに対して割り当てられた仮想空間内で興味のある特定の項目を見つけることができる．

そこで，ファイル内の VAM レコードをユーザに対して検索してくることは，ファイル全体とそのディレクトリとに仮想空間を割り当てることと等価に

なる．しかしながら，割り当てたすべての空間が物理記憶に入っているわけではない．そこで，ユーザは，VAM ファイル中のある部分の VAM レコードを選んでアクセスしてよく，その結果，必要な部分の VAM レコードだけを物理記憶の中に転送してくれればよい．

内容表のほかに必要となるレコードの一部だけを物理記憶にもってくればよいので，仮想アクセス法が効率が良いのはあきらかである．

このアクセス法は，ページング操作とよく似ている．仮想空間のページが物理記憶の中に置かれていなければ，そのページに対するページ表のエントリは，“物理記憶にはない”と印がつけられる．そこで，このページ表のエントリの番地は，補助記憶上のページの番地となる．この番地を使って，ページングルーチンは，補助記憶に対する物理入出力操作を開始する．同様の技法を，ファイルに対して二次記憶機器上に存在するページに適用するように拡張する．VAM レコードのページに対して仮想空間を割り当てるときには，ページ表のエントリをページに対して割り当てるが，これらのエントリは，“仮想空間にない”という印をつけ，このエントリに与えられている番地は，二次記憶機器上の番地である．“物理記憶にない”と印をつけたページを参照したときには，ページフォールトの割り込みが生じて，補助記憶（すなわち，ページング機器）からの入出力操作が行なわれる．“仮想空間にない”という印がついているページを参照したときには，やはりページフォールトの割り込みが生ずる．この場合，生じてくるページング操作は，ページング機器でなく二次記憶機器から読み込む操作である．それゆえ，ページング操作と仮想アクセス法との類似点は明きらかである．

仮想アクセス法の操作を，ここで少し詳しく述べよう．VAM ファイルの VAM レコードは可変長であるけれども，ファイルディレクトリの中にはレコードの長さに関する指示は何もない．特に，VAM レコードの TOC もやはり可変長である．一方，ページの大きさとトラックの大きさは固定しているので，物理レコードは固定長である．したがって，可変長の VAM レコードは，標準の固定長の形式に適合させなければならない．固定長形式の選択は，それぞれの仮想記憶システムで異なる．物理レコードの大きさは非仮想記憶システムではもっと柔軟にできるが，それとは異なり，仮想記憶システムに対して選んだ物理レコードの大きさは，トラックの大きさに加えてページの大きさにも関係してこなければならない．普通，この形式の大きさは，ページの大きさ（あるいは，半ページを参照する機能が使えるならば半ページの大きさ）の固定した何倍かになる．この議論では，物理レコードをトラックとよぼう．

これらのレコードを固定長形式に合わせる方法は，TOC に対する TOC を定義すること，すなわち，本質的には，TOC を多重レベルのディレクトリの形式に構成することである．したがって，VAM レコードの TOC は，TOC の第一レベルで固定長の部分だけを指している．そこで，TOC から始まって，レコードの中の特定の項目をアクセスするのは，多重ステップの操作になる．

VAM レコードの残りは，二次記憶機器の番地空間で TOC トラックに続く物理トラック上にある．そこで，VAM レコードの先の物理(二次)番地がわかり，データ要素の(レコードの先頭に対する)論理番地がわかれば，データ要素を含むトラックの番地がわかる．

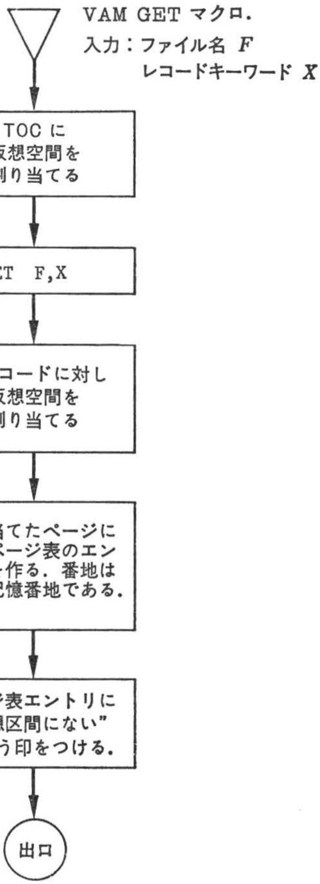


図 5.25 VAM GET マクロ

VAM GET マクロのステップの概要を，**図 5.25** に示す．最初に，VAM GET マクロは，このファイルのレコードに対する TOC を含むのに十分な仮想空間を割り当てるために，システムルーチンを呼び出す．そして，レコードの長さを含む TOC を，仮想空間内にもってくる．VAM TOC にアクセスするために，前に述べた正規の GET マクロを使ってもよい．レコード A の TOC に対して割り当てられた仮想空間を使って，GET F, A 操作でレコード A に対する TOC を仮想空間内に持ってくると考える．これは，**図 5.26** の操作(1)である．

次に，全体のレコードを含むのに十分な仮想空間を割り当て，この仮想空間に対するページ表エントリを作る．これらは，図 5.26 の(2)と(3)である．

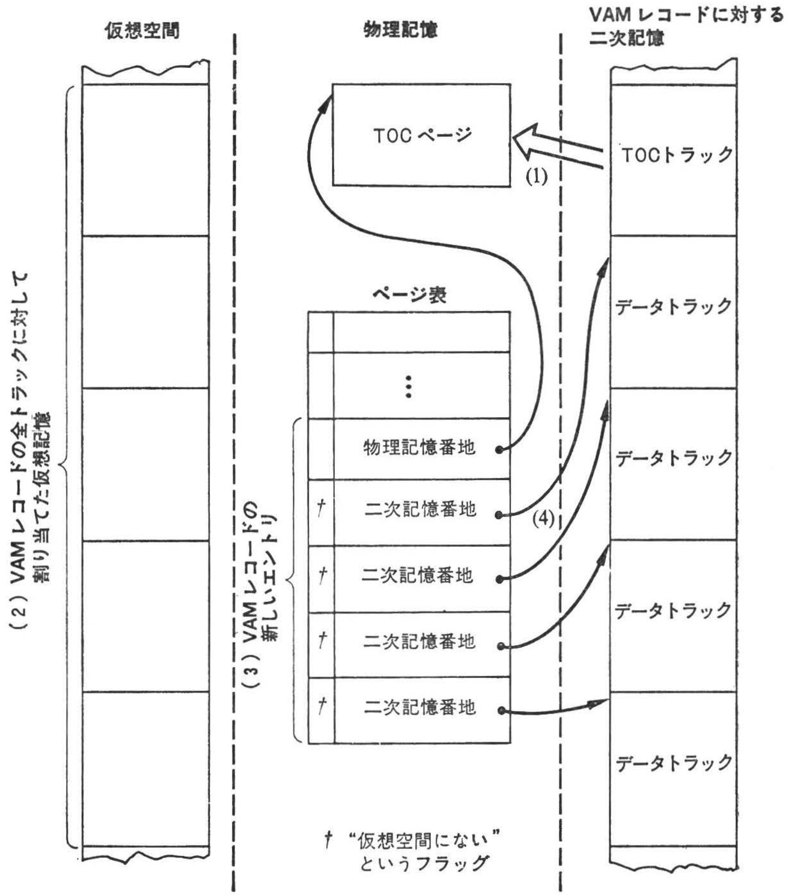


図 5.26　VAM GET の操作

ページ表のエントリは，“仮想空間にない”というフラグを持っている．ページの二次番地をレコードの最初の二次番地から計算して，ページ表に入れる．これは，図 5.26 の操作(4)である．

二次記憶上のファイルとレコードが変換表を更新しページングに似た入出力操作を利用することにより，“論理的に”仮想空間の中に移されるだけでなく，記憶の階層の一部としてページのための補助記憶であろうとファイルやレコードに対する二次記憶であろうと(物理記憶以外の)すべての記憶を考えれば，更にいくつかのステップを進めることもできることに注意するのは興味深い．物理(主)記憶は，プログラムを実行する記憶である．階層は，情報のオンラインの格納と検索のためにある．階層の中で情報を格納し検索するために“統一的な”入出力操作を提供するためには，これらの入出力操作は十分“知能的”でなければなく，その結果，プログラムのページの様にもっと直ぐに必要となる情報は高速の検索と格納をするために高性能の記憶に格納し，ジャーナルのテープの様にあまり頻繁には使わない情報は，少ししかアクセスしないように低性能の記憶に格納する．恐らく，その様な階層が，前に概略を示した単層記憶の概念に対する第一歩になるであろう．

5.2.2.5 仮想空間の節約

仮想記憶システムにおいて仮想空間を使うことは，システムの物理記憶を節約しようという試みと見ることもできる．物理記憶上でアクティブなタスクを処理しているときには，任意の時点で絶対的に必要となる(通常 1 ないし 2 ブロックの)物理記憶だけをタスクに割り当てて，高価な物理記憶を倹約している．タスクの残りの部分(すなわち，アクティブの度合いが少ない部分)に対しては，高価格の物理記憶の代わりに低価格の補助記憶を割り当てる．

ある時点では，各タスクの全仮想空間を含んでいる補助記憶でさえ，その価格が大き過ぎることになるであろう．物理記憶を倹約するために使ったものと同じ技法を，仮想空間を倹約するために再度使うことができる．すなわち，仮想空間の中でアクティブに動いているタスクの部分だけに，補助記憶を割り当てる．勿論，ここでいう“アクティブ”の操作的定義は，物理記憶に存在させておくタスクの部分を決めるための定義† に比べて，ゆるやかなものにしなければならない．仮想空間の残りの部分は，何かもっと低価格の大容量記憶機器上に置いておく．勿論，これを論理的に拡張すると，記憶機器の階層ができる

† 訳注 ページ置き換え則として，最長未使用(LRU)，先入先出(FIFO)，ワーキングセットを利用したものなどいくつかの方法が使われている．

が，階層内の各機器は，一つ上のレベルの機器に比べて，大容量で低速，かつ，(情報の単位あたりが)低価格になっている．プログラムとデータは，最上層で何分の一マイクロ秒から最下層で数年の範囲にわたる期間使用するということで，階層内を上下するであろう．このような拡張も，やはり，単層記憶の概念である．ここでは，階層は，記憶機器の構成をあらわす．しかし，ユーザやユーザタスクに関する限り，この階層全体がその仮想空間になる．換言すれば，階層はユーザには見えず，機器が支援してくれる記憶は単層に見える．

A.　記憶節約のための再入可能型プログラムの利用

ユーザプログラムを再入可能型に書く技法については，前の章で論じた．この技法では，一人のユーザが他のユーザが使っているあるページ(すなわち，そのユーザはそれに対して物理記憶を割り当ててある)を全く同時に要求したときにのみ，物理記憶の節約ができる．このような使い方を実現するのは難しいので，再入可能型プログラミングの技法によって物理記憶を節約することを達成するのは困難である．しかしながら，そのプログラムのすべてのユーザに対してただ一つの写しを補助記憶に置いておくだけで良いので，この技法により補助記憶を極めて最適に使うことができる．

B.　記憶節約のための動的呼び出しの利用

補助記憶のもう一つの節約は，**動的呼び出し**(dynamic call)である．ユーザタスクで一つのプログラム A を実行していて，そこで，もう一つのプログラム B の処理が必要になったと仮定する．これらのプログラムは2つとも同じ仮想空間にロードして，お互に通信できるようにしなければならない．伝統的には，これらのプログラムは，タスクを実行する前，すなわち，プログラム A の実行が始まる前に，同一の仮想空間内にロードして結合するようにしてきた．(このローディングと結合の過程は，この章の前半部で考えていたように，静的な再配置がその特徴である)．それゆえ，タスクの仮想空間は，そのタスクの処理の間ずっと，両方のプログラムを入れるのに十分大きなものとしておかねばならない．

実際の B への呼び出しが生ずるまでプログラム B を仮想空間にロードしなければ，タスクに対する仮想空間の節約になることはあきらかである．システムの CALL マクロは，プログラム B のローディングと結合の機能を実行し，そこに制御を移す．したがって，CALL マクロは，**動的結合ローダ**(dynamic linking loader)と呼ぶローダを利用する．次節では，動的ローダ，CALL マク

ロの利用，新しいローディングと結合の過程について述べよう．

5.2.2.6 動的結合

A. 動的結合ローダの機能

ローダは，次のような機能を実行する．

1. プログラムに必要な仮想空間を割り当てる．
2. プログラムを仮想空間に移す．
3. プログラムの内部的再配置と外部記号のすべてを解決する．

第一の機能を実行するときには，ローダは最初に(a)ロードするプログラムの大きさを決定し，(b)プログラムに対して補助記憶のトラックを割り当てるためにシステムの表を参照し，(c)プログラムを要求したタスクのページ表に対しこのプログラムのエントリを作らなければならない．

第二の機能は，プログラムを仮想空間に移す物理入出力操作を伴なう．

第三の機能に対しては，番地の解決のために必要なデータを仮想空間にもってきてローダが使えるようにするために，入出力操作を行なわなければならない．そうすると，プログラムの内部的な再配置の記号の解決を，仮想空間内で行なうことができる．しかし，外部記号の解決のためには，物理入出力操作を何回か行なうことになり，ローダをもう一度呼ぶことになるかもしれない．

B. 番地参照の解決

ここで，番地の解決の過程を見てみよう．勿論，この過程は，第1章で論じてきた．しかしながら，第1章に示した問題と関連づけながら，仮想記憶に対するものをもう一度以下で論じてみよう．

プログラムの各々の命令は，命令のオペランドを指定する仮想番地を潜在的に含んでいる．プログラムをアセンブルしたりコンパイルするときには，プログラムはどの仮想空間にも入っていない．したがって，プログラムに含まれている番地は，完全に指定が行なわれているわけではない．番地は，プログラムの先頭からの相対的な偏差として示されており，相対番地と呼んでいる(1.4.4節参照)．プログラムを仮想空間にロードするときには，プログラムに含まれている番地を完全に決めなければならない．一般に，ローダがこれを行なうために，プログラムの先頭に対する絶対的な仮想番地を，プログラムに含まれる相対番地にそれぞれ加える．そのような過程を，**番地解決**(address resolution)と呼ぶ．

ローダが解決すべき番地の数を減らす方法が，いくつかある．一つは，ベースレジスタ番地付けの技法である．プログラムの実行の最初の2, 3の命令で，プログラムの開始番地とデータ領域の先頭番地をベースレジスタに入れる．プログラムの残りの部分を通して，ベースレジスタにある番地を使う命令が，番地を陽に含んでいる命令に置き換る．ベースレジスタを介して番地を陰に参照し，また，プログラムの主要部分で番地を陽に使うことをやめることにより，プログラムは，2つ以上の仮想空間に存在することができると考えてよい．そのようなプログラムは，いくつかの仮想空間で**共用プログラム** (shared program)として使うことができる．

上述の技法を使えば，プログラムが仮想空間で再配置されたとき，プログラムの内部再配置形記号の番地を解決する問題を，簡単化することができる．

番地解決におけるもっと難しい仕事は，一つのプログラムAで使う番地が別のプログラムBで定義されているときに起る．AとBとが別々にコンパイルされたリアセンブルされているならば，プログラムAの言語処理系は，Bの内部で定義されている記号の値に関する知識は何ももっていない†．これらの記号はどの仮想空間でも定義されていない事実とは別に，プログラムBに対する相対的な値すらわからない．したがって，Aの言語処理系は，これらの外部記号を，英数字による名前の形式のままにしておく．これらの記号の値は，プログラムが仮想空間にロードされる時点で決めなければならない．そのため，言語処理系の出力(すなわち，目的モジュール)は，プログラムそのもの(すなわち，プログラム自身を構成する命令や定数)と，番地指定をする情報に関する2つのディクショナリ，すなわち，内部再配置記号ディクショナリと外部記号ディクショナリとから構成される．あきらかに，記号の値を計算するためにローダが使う情報は，前述のディクショナリの中に含まれている．

第一のディクショナリは，番地を含んでいるプログラム内の場所を指定する．ローダは，プログラムが仮想空間内にロードされてきたときに，これらの場所を修飾するためにこのディクショナリを使う．

第二のディクショナリは，2つの型の情報を含んでいる(1.3.2.2節も参照のこと)．第一の型は，**外部定義記号** (externally defined symbols; **REFS**)と呼ぶ外部記号名の集合である．これらは，どこか外のプログラムで定義されプログラムA内で使っている名前である．これらの英数字名は，言語処理系

† 訳注　ALGOLやPASCALのようなブロック構造をもつ言語で，ブロックの入れ子構造で決まる環境の情報を言語処理系に与えてコンパイルをする分割形コンパイラの試みが行なわれている．

により **REF 表**に登録される．一般に，A 内には一つの外部定義記号を参照する場所が，多数あるかもしれない．したがって，目的モジュールのプログラム自身では，外部参照をしている各点において，REF 表のエントリへのポインタをもっている．REF 表のエントリがその番地が必要となるプログラム内の各場所に対するポインタをもっていてもよいけれども，通常そのような技法は使わない．どちらの場合にも，ローダは，外部参照の適切な定義をプログラムの中に置けるような機能がなければならない．番地を必要とするプログラムの語(またはバイト)と名前自身との結合は，存在しなければならない．

番地の情報の第二の型は，**記号定義**(symbol definition; **DEFS**) とよぶエントリポインタの集合である．プログラム A の中で，どこかほかのプログラムから参照してよい記号の場所がある．言語処理系は，英数字名であらわされている DEF と，プログラム A に対する相対番地であるそれらの値との対応

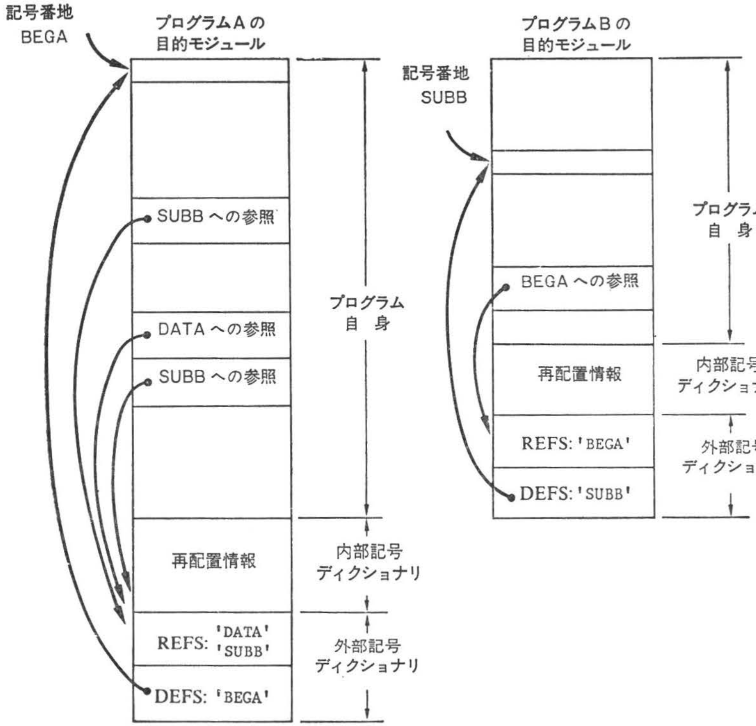


図 5.27 目的モジュールに含まれる番地指定の情報

表を作成する．この表を，**DEF 表**という．

REF および DEF 表の結合を陽にあらわした2つの目的モジュールを，図5.27に示す．

プログラムが仮想空間内にロードされたあとは，その内部的参照はすべて決まってしまったので，（内部番地に対する）再配置の情報は捨ててもよい．次に，REFS を解決しなければならない．プログラム B の DEFS，すなわち，プログラム A の REFS がその仮想番地を割り当ててもらわないと，ローダはプログラム A のローディングを完了することができない．ローダはプログラム B に空間を割り当て，プログラム A のローディングを終了させる前に，それを仮想空間にロードする．プログラム A 内の REFS は，その仮想番地を与えられ，次いで，REFS の表を捨てる．後でロードされるプログラムがプログラム A 内の場所への REF をもっているかもしれないので，プログラム A の DEF 表はとっておかなければならない．さらに，DEF 表は，今や，相対番地をもっている必要はない．DEF 表の各記号は，今や，仮想番地の値をもっているので，表を更新して，DEF 記号とそれらの仮想番地との対応をもつようにする．

潜在的には，一つのタスクに対してロードしてきたプログラムと同じ数だけの DEF 表がある．表を統合して解決をするために，これらの表をそのタスクの**タスクディクショナリ**（task dictionary；**TDY**）と呼ぶ領域に移す．新しいプログラムがタスクに対してロードされる度に，新しい REF ごとにそのタスクの TDY をさがす．新しいプログラムの REF がすべて TDY で見つかるならば，新しいプログラムのロードをする際に REFS の解決をするために動的結合ローダを再帰的に呼び出す必要はなくなるであろう．

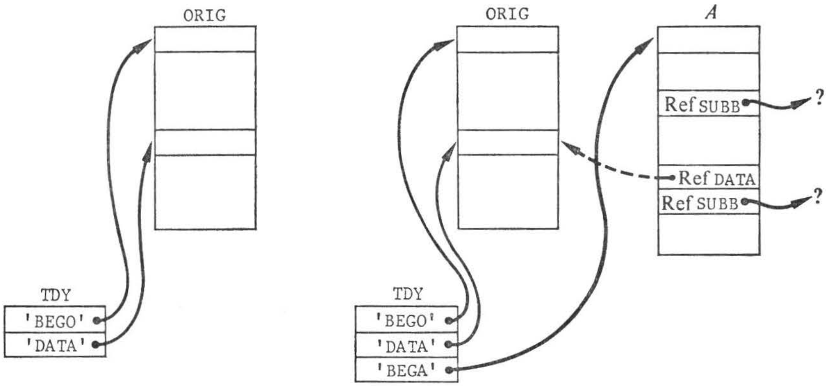
C.　ローディングと結合の操作

タスクに対してプログラムを仮想空間にロードしてくるために，ローダは次の5つのステップを実行しなければならない．ここで，いくつかのステップは，他のステップが実行される前に必ずしも完了しているとは限らない．

1. 仮想空間をプログラムに対して割り当てる．
2. プログラムのテキスト（すなわち，プログラム自身）を仮想空間にもってくる．
3. プログラムのテキストの相対番地をすべて更新する．内部再配置記号と外部記号ディクショナリにある再配置の情報は，このステップの間使えなければならない．

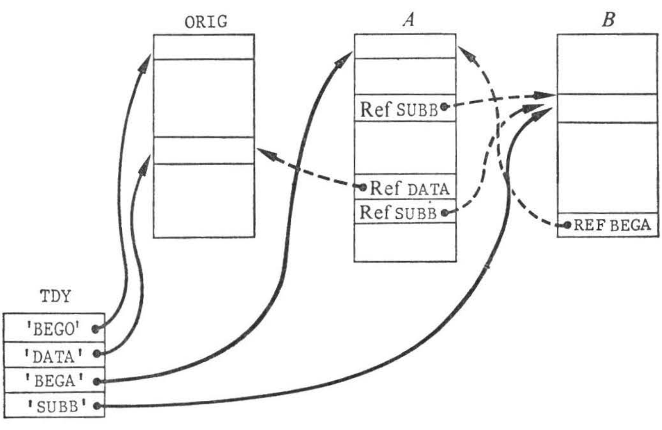
4. DEF 表の各記号の番地を完了して，この表を更新する．ここで，DEF 表をこのタスクの TDY に統合して，このプログラムに対する REF を含む別のプログラムをロードしてくる間にローダが参照できるようにする．

5. REF 表の各エントリを定義する．この処理では，



(a) プログラム A がロードされる前

(b) プログラム A をローディング中は SUBB への参照は未定義である．



(c) プログラム B がロードされ，他のプログラムと結合した後

図 5.28 外部記号の解決

a)　各 REF 記号に対し，TDY を探索すること，

b)　TDY にそれが見つからなければ，その記号に対しユーザのファイルカタログを探索すること，

c)　ユーザファイルの定義領域にそれが見つからなければ，Syscat の中でその記号を探索すること，

を行なう．

b)および c)の場合，REF は未だロードされていないプログラムで見つかる．したがって，REF は，対応するプログラムがロードされるまで定義はできない．その結果，その記号を含んでいるプログラムを見つけなければならず，このステップを終了するまでには，ローダを再帰的に呼び出さねばならない．記号が定義されたときには，プログラム内にあるそれへの参照はすべて完了する．最後に，これが済めば，REF 表は捨ててもよい．

図5.28では，図5.27のプログラム A の各々の REF が A がロードされるとともにどのように解決されてゆくのかを例として示している．A がロードされる前には，図5.28(a) に示すように，このタスクの仮想空間にプログラム ORIG が存在する．プログラム A をロードしている間，図5.28(b)に示すように前にロードしたプログラムの中で定義されているすべての外部記号を含んでいる TDY の中で，REF SUBB は見つからない．ファイルカタログの探索をすると，SUBB はプログラム B の中で定義していることがわかる．それゆえ，プログラム B をロードして，プログラム B の更新した DEF 表を TDY につけ加える．A 内の番地は，図5.28(c)に示すように，今や完全に決めることができる．

D. 仮想アクセス法を使ったローディングと結合

前節に示したように，ローダは入出力操作を良く使うユーザである．これらの入出力操作の間，機械の利用を出来るだけ効率的に行なうために，ローダは仮想アクセス法 (VAM) を使用したい．仮想アクセス法を採用したとき，データはそれが使われるときにだけ，一時には1ページづつ物理的に移動する．この場合，プログラムのページは，それを実行中にアクセスしたときにだけ仮想空間にもってくる．すなわち，プログラムはそれが完全にロードされ結合される前に，実行することを“強制される”．しかしながら，ローダは，プログラムそのものを二次記憶機器から持ってきてからプログラムの番地の解決をしなければならない．これは，ローダの機能が分散しているので，ページを二次記

憶からもってくるとき，プログラムのページを一時には一つづつローダが処理していることを意味している．

ローダが呼ばれて一時には1ページづつロードするならば，プログラムの全ページがロードされるまで，仮想空間で番地を決定するすべての情報をローダは保持していなければならない．各ページに対する番地決定の情報は，ページがロードされるとき物理記憶の中で使用可能でなければならない．

そこで，2つの独立した操作が必要となる．一つは，プログラムのローディングの準備をするものである．この操作は**プログラムロード**(program load)操作とよばれ，各ページをロードするごとに一回づつよばれる．この操作の間，プログラム自身が動くことはない．

第二のロード操作は**ページロード** (page load) 操作であるが，それは，ロードすべきプログラムの各ページの最初にアクセスされたときに実行される．ページロード操作を呼び出すあきらかな場合は，ページフォールトに応答するときである．

VAM を使うプログラムロード操作は，次のようなマクロとして実現することができる．プログラムの名前をパラメータとして，この操作は，VAM GET を呼び出す．VAM GET は，このプログラムに割り当てられた仮想番地，プログラムの長さ，および，内容表(TOC)の仮想番地を戻す．目的モジュールの(VAM)レコードの場合，TOC は目的モジュールの (DEF 表や REF 表も含めた) 再配置仕様の情報の番地をもっている．そこで，この操作は DEF や REF 表を TDY に移して，後のページロード操作でも参照できるようにする．この情報にアクセスするや否や，ページフォールトが生じ，VAM ページングルーチンを呼び出すことになる．これにより，プログラムの再配置仕様の情報を含んだページを物理記憶にもってくる．

VAM の呼び出しでは，プログラムがロードされる仮想空間を与えるから，ローダはプログラム内の各相対番地とその仮想番地との対応を作ることができる．特に，ローダはプログラムに対する DEF 表を完成することができる．REF 表は，REFS が必要になったとき，すなわち，ページロード操作を呼び出した時に完成していればよい．

最後に，(VAM)ページロード操作は，ページがデータやプログラムを含んでいるかどうかを知らなければならない．この情報は，ページ表に残っている．‘物理記憶にない’あるいは‘仮想空間にない’というビットに加えて，プログラムセクションを含んでいる各ページ† に対しては，‘ローダによって未処理’のビットがある [† 注は次頁を参照].

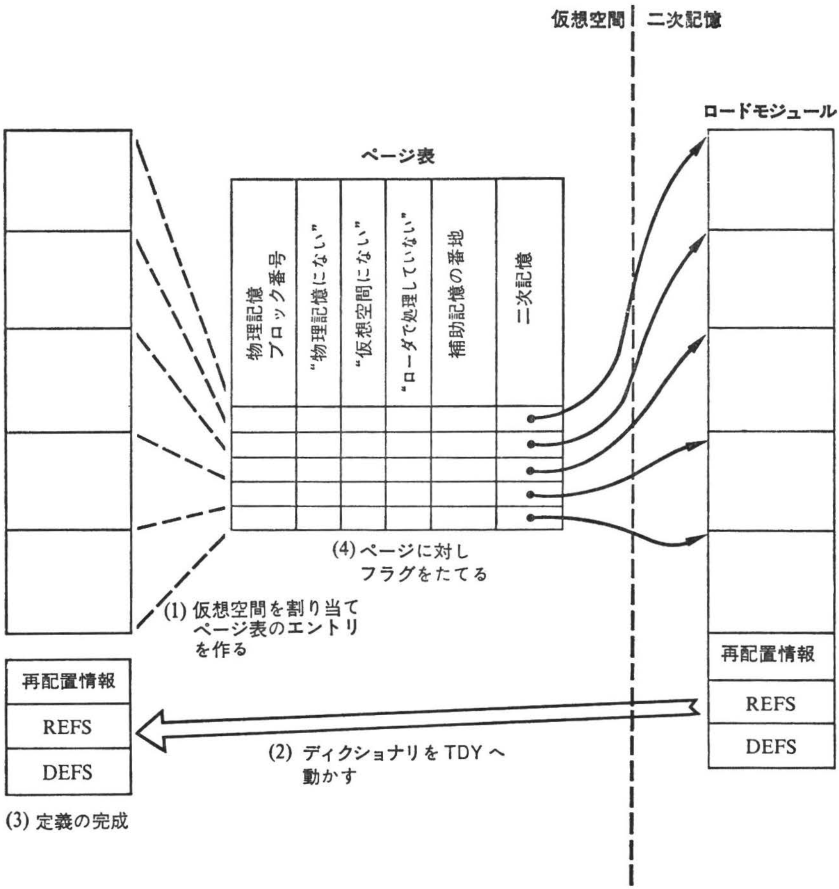


図 5.29　VAMを使ったローディングと結合の操作

ロード操作の概要を図5.29に示す.

E.　ページローダ

プログラムの実行中にページフォールトが生ずると，ページフォールト割り込み応答ルーチンが，ページ表の'仮想空間にはない'というビットを調べる．それがセットされていれば，ページングルーチンはページに対する物理入出力

前頁の注：
† プログラムは，少なくとも一部分は番地常数を一つも含まないように書けるので，プログラム自身の中にあるすべてのページがローディング操作を必要としているわけではない．たとえば，再入可能型プログラムは，修飾可能なデータの番地は含んでいない．

を開始する．物理的読み込みの操作が終了したときには，ルーチンはそのページに対する‘ローダで未処理’のビットを調べる．それがセットされていれば，ローディングの処理を始めなければならない．プログラムのページのローディングを扱うシステムプログラムを，**ページローダ**(page loader)という．

ページローダは，TDY の中のこのページに対応する再配置の情報を見つける．再配置の情報の表にある指定の語各々に対し，番地計算を行なう．どこかの番地が REF の定義を必要とするならば，その REF に対して TDY を探索する．この探索が失敗すれば，記号の定義に対し，第一にユーザカタログ，第

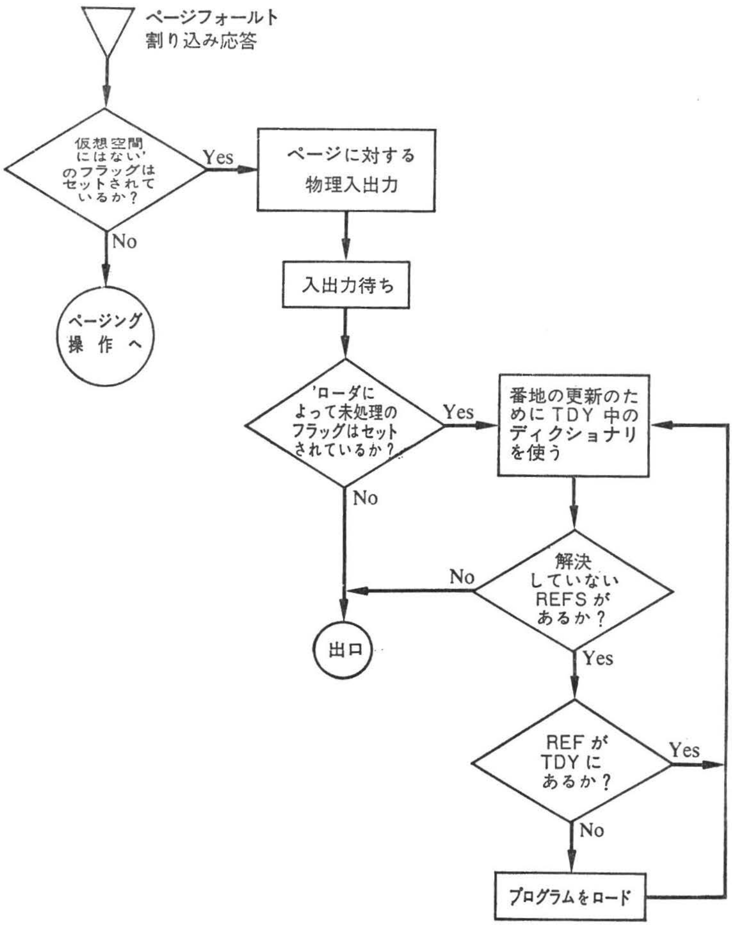


図 5.30 ページローダ

ニにシステムカタログを探索する．もしそれが見つかれば，REF を解決するために，この記号を含んでいるプログラムをロードしなければならない．

ページローダの概要を，**図 5.30** に示す．

5.2.2.7　多重タスキング

多重タスキングの概念は，前節でその動機づけを述べてきた．実行中のタスクを協調させるために同期と相互ロックの機構が必要であることも，共用資源上のシステムデッドロックと関連してそこで示してきた．ここでは，それらの議論を繰り返すつもりはない．この節では，タスクの生成と消滅の過程を論じよう．その考え方がなければ，多重タスキングなどは存在しない．

一つのタスクが複数個の論理的機能を実行しなければならないときには，新しいタスクを作りたいと思う．これらの論理的機能をいくつか実行するために別の(副)タスクを作れば，いくつかの機能が並行的に実行されるので，必要な全体の時間が減少するだろうと期待される．タスクと副タスクは，システムの資源を競争しながら単一の処理装置上でも並行に実行してよい．たとえば，一つのタスクが処理装置をとり，もう一方のタスクが入出力操作を待っているという結果になる．新しい副タスクに対して TCB を作るときの新しいタスクの開始点をあらわす仮想番地だけが必要となる．新しいタスクに対するページ表，ファイルの機密保護やアクセス制御の情報などのデータは，元のタスクの TCB から直接写してきても良い．実際，副タスクはダミーの TCB だけを持っていて，元のタスクの TCB に対するポインタ以外に何も持っていなくてもよい．

副タスクを作成するための CREATE マクロは，副タスクの開始場所をどのように指定するかによって，次のいくつかの方法で処理される．これは，次の3つの方法のどれかで指定してよい．すなわち，(1) プログラムのリテラル名，(2) システムプログラムの SVC 番号，(3) プログラムの仮想番地，である．第一の場合，CREATE マクロによりプログラムの名前を Syscat の中で探索し，それを見つけ，そこに制御を移す．各々の場合，CREATE マクロが制御の引き渡しを行なうために，ルーチンからの戻りを受ける小さいプログラムを作り，そのプログラムに SVC か飛び越し命令を挿入し，それから作成したプログラムに制御を移す．

図 5.31 には，CREATE マクロの処理の概略を示す．副タスクに対する開始点を指定する3つの一般的方法を，このマクロで翻訳しており，副タスクの開始をあらわすプログラムを示している．副タスクの処理は，ここで示している

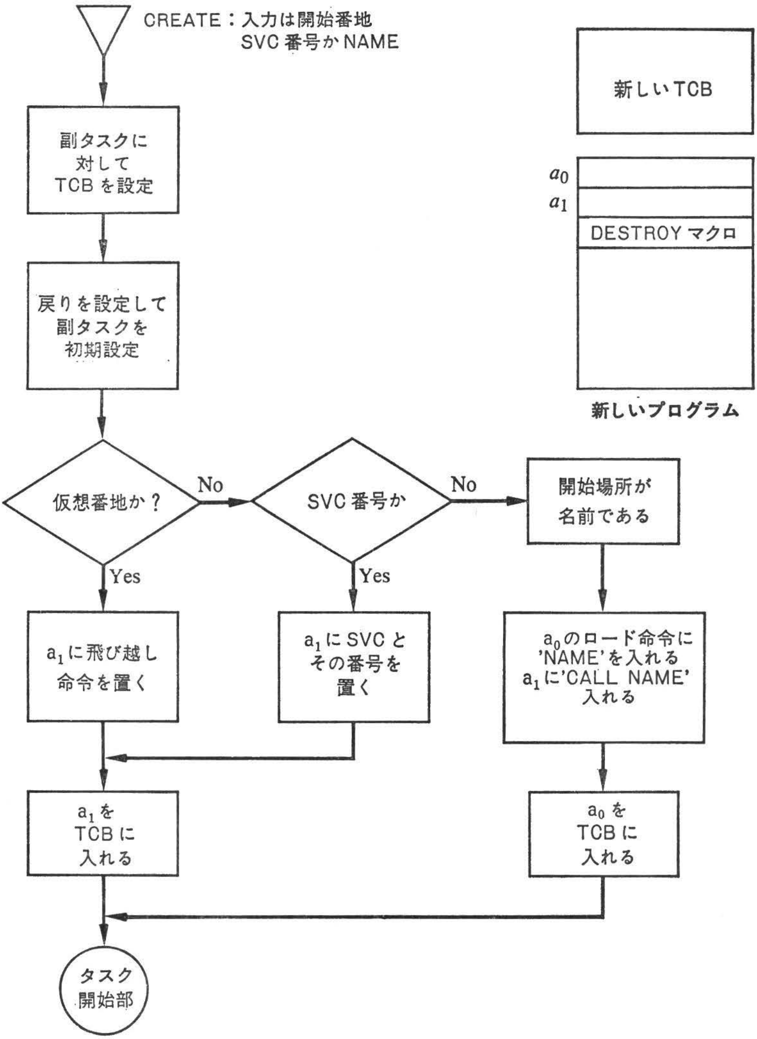


図 5.31　CREATE マクロ

サブルーチンの中で行なわれ，副タスクに対するプログラムは，サブルーチンへの飛び越し命令と戻り場所から構成される．

副タスクの実行をした後，DESTROY マクロを出してこれをシステムから除くことができる．DESTROY マクロは，CREATE マクロと似たような方式で働

くので，ここで詳しく述べることはしない．

5.2.3 要　約

一般化した仮想記憶システムの提示を，一括処理システムの提示と同じ方式によって行なった．その方法では，最初にオペレーティング・システムの常駐部を調べ，次に，PIOCS と LIOCS に進み，最後にファイルシステムを述べた．チャネル，機器，入出力操作のような物理的操作や資源を扱う基本的機能から見ると，一括処理システム，多重プログラミングシステム，仮想記憶システムの間にはかなりの相似点がある．オンラインシステムに対しては，それが一括処理であろうと，多重プログラミングであろうと，仮想記憶システムであろうと，論理入出力操作の集合と集中型のファイルシステムの必要性は，普遍的である．したがって，立派な機能をもった仮想記憶システムが，オンラインであろうとオフラインであろうと，一括処理システムも支援できるということは，驚くにあたらない．仮想記憶システムを特徴づけることは，**会話型処理**(interactive processing)を提供できる能力である．会話型処理システムでは，速い応答時間に非常に大きな努力が向けられる．前に述べたように，応答時間とは，出力の要求から出力の到着までに経過する時間を意味していた．速い応答を実現するためには，時間刻みと順番まわしの手続きとを，ユーザジョブのスケジュールで使う．順番をとり，システムから刻み時間を受け取ると，競争しているジョブはある程度処理が進んでいくらか出力を出し，その結果何らかの応答をする．あるジョブの順番で進むとき，仮想記憶を記憶の中で再配置しなければならず，そのデータやルーチンに対して空間を割り当て，それが使うデータやルーチンを回復し，そのルーチンを動的にロードし結合する．仮想記憶システムは，プログラムを繰り返し再配置させ，動的に空間を割り当て，二次記憶上のデータやルーチンに(VAM を介して)効率よくアクセスし，実行中にルーチンをロードし結合する能力を持っいるので，これらの要求を満たすことができる．応答時間を改良する方法は，ほかにもある．たとえば，元のジョブを，もっと改善したタスクに構成することができる．したがって，多重タスキングの概念を導入したわけである．改善したタスクは，元のジョブに比べてシステムの処理量を管理するのを容易にしてくれるので，多重タスキングによりシステムの利用率も改善される．良い応答時間と処理量をもっていれば，資源の大部分を他のユーザと共用しているけれども個々のユーザはすべてのシステム資源を自分の好きなように使えると考えているようなところでも，会話型仮想記憶システムは多くのユーザを処理することができる．そのようなシステム

を，しばしば**時分割システム**(time-sharing system)という.

5.3 あ と が き

仮想空間の議論は，[10] と [18] に概説してある見方に基づいている. 仮想空間の実現の特定の例は，[14], [15], [24], [27], [30] および [31] からもってきた. 単層記憶の概念は，[16] から始まり，セグメンテーションの概念は Burroughs 社の B 5500 に負っている. これらの概念は，[6] で一般化している.

一般化した仮想記憶システムは，[5], [19], [23], [28] および [32] に見られる流行している考え方や共通の取り組み方を使って設計される. 一般化した仮想記憶システムの節の最初の原稿は，(まえがきにも述べた) 1970 年の**多重プログラミングシステムの設計原理**に対する夏のプログラムに対して書かれ，それ以後 [21] で扱っていた.

OS/VS 2 Release 1 や Release 2 のような最近の IBM 370 のオペレーティング・システムに導入され，仮想記憶システムは一層の推進がはかられるようになった. OS/VS 2 Release 1 や Release 2 は，第 4 章で論じてきた OS/360 MFT や MVT に置き換わろうと意図しているものである. Release 1 と Release 2 の大きな違いは，仮想空間の支援の部分である. Release 1 では，唯一つの仮想空間しかない. Release 2 では，各ユーザに対し，一つの仮想空間が存在できる. 仮想空間の多重度については，この章の前半部で論じている. この章で扱っている仮想記憶システムは，これら大規模なシステムの入門として使える. ,たとえば，ファイルカタログの方式は，Release 1 および 2 で使っているものを簡略化したものである. 仮想アクセス法(VAM)は，これら 2 つの Release の VSAM および VIO の導入部となる. OS/VS 2 Release 1 および Release 2 の参考文献としては，[28] から始めると良い.

紙面が限られているので，この章では，ページングアルゴリズム，プログラムに対するページングの影響，要求ページングおよびその他のページングの方式，ページ化記憶の記憶の断片などの話題のいくつかについて論ずることは出来なかった. しかしながら，ページングアルゴリズムについては [2]，プログラムに対するページングの影響については [4]，要求ページングについては [7], [17], [29]，ページング方式については，[9] と [26]，ページ化記憶の記憶の断片については [25]，ページングに対する補助記憶については [8] の中に，いくつか興味のある参考文献を読者は見つけられるであろう.

この章では，演習問題をつけなかった. しかし，この章で概略を示した一般化

した仮想記憶システムは，いくつかのシステムの要素の設計や実現の習得の基礎として使える．もっと野心的な試みをするならば，この研究から得られるシステムの要素を使って，仮想記憶計算機上に完全なシステムを開発するのもよい．

参　考　文　献

1. M. A. Auslander, and J. F. Jaffe. "Functional Structure of IBM Virtual Storage Operating Systems—Part I: Influences of Dynamic Address Translation on Operating System Technology." *IBM Systems Journal* **12,** 4 (1973): 368–381.
2. L. A. Belady. "A Study of Replacement Algorithms for a Virtual Storage Computer." *IBM Systems Journal* **5,** 2 (1966): 78–101.
3. J. P. Birch. "Functional Structures of IBM Virtual Storage Operating Systems—Part III: Architecture and Design of DOS/VS." *IBM Systems Journal* **12,** 4 (1973): 401–411.
4. B. S. Braun and F. G. Gustavson. "Program Behavior in a Paging Environment." *Proc. AFIPS 1968 FJCC* **33,** 2 (1968): 1019–1032.
5. F. J. Corbató and V. A. Vyssotsky. "Introduction and Overview of the MULTICS System." *Proc. AFIPS 1965 FJCC* **27,** 1 (1965): 185–196.
6. R. C. Daley and J. B. Dennis. "Virtual Memory, Processes and Sharing in MULTICS." *Comm. ACM* **11,** 5 (May 1968): 306–322.
7. W. M. Demeis and N. Weizer. "Measurement and Analysis of a Demand Paging Time Sharing System." *Proc. of 24th National Conference of ACM*, 1969, pp. 201–216.
8. P. J. Denning. *Queuing Models for File Memory Operation*, MAC-TR-21 (M.S. Thesis), Project MAC, MIT, October 1965.
9. P. J. Denning. "The Working Set Model for Program Behavior." *Comm. ACM* **11, 5** (May 1968): 323–333.
10. J. B. Dennis. "Segmentation and Design of Multiprogrammed Computer Systems." *Journal of the ACM* **12,** 4 (October 1965): 589–602.
11. H. Hellerman. "Some Principles of Time-Sharing Scheduler Strategies." *IBM Systems Journal* **8,** 2 (1969): 94–117.
12. D. Howarth. "The ATLAS Supervisor Program, Introduction to System Programming." *System Programming*. Edited by P. Wegner, New York: Academic Press, 1964, pp. 227–238.
13. D. K. Hsiao et al. *A Manual for RCA SPECTRA 70/46 TSOS Subsystem Writers*. Technical Report, Moore School of E.E., University of Pennsylvania, June 1969.
14. "IBM System/360 Model 67 Functional Characteristics," File No. S360-01, Form A27-2719-0, 1967.
15. "IBM System/370 Principles of Operation," GA22-7000-3, Fourth Edition, January 1973.
16. T. Kilburn, D. B. G. Edwards, and F. H. Sumner. "One-Level Storage System." *IRE Transaction on Electronic Computers*, April 1962.
17. C. J. Kuehner and B. Randell. "Demand Paging in Perspective." *Proc. AFIPS 1968 FJCC* **33,** 2 (1968): 1011–1017.

18. W. C. McGee. "On Dynamic Program Relocation." *IBM Systems Journal* 4, 3 (1965): 184–196.

19. G. H. Mealy, B. I. Witt, and W. A. Clark. "The Functional Structure of OS/360—Part I: Introductory Survey; Part II: Job and Task Management; Part III: Data Management." *IBM Systems Journal* **5,** 1 (1966): 2–51.

20. D. Morris, F. H. Sumner, and M. T. Wyld. "An Appraisal of the ALTAS Supervisor." *Proc. of 22nd National Conference of ACM*, 1967, pp. 67–75.

21. A. Noetzel. "The Design of a Meta-System for Measurement and Simulation of Time-sharing Computers." *Ph.D Dissertation*, University of Pennsylvania, 1970.

22. R. W. O'Neill. "Experience Using a Time-Shared Multiprogramming System with Dynamic Address Relocation Hardware." *Proc. AFIPS 1967 SJCC* **30,** (1967): 611–621.

23. G. Oppenheimer and N. Weizer. "Resource Management for a Medium Scale Time-Sharing Operating System." *Comm. ACM* **11,** 5 (May 1968): 313–322.

24. E. I. Organick. *The Multics System: An Examination of Its Structure.* Cambridge, Mass.: The MIT Press, 1972.

25. B. Randell. "A Note on Storage Fragmentation and Program Segmentation," *Comm. ACM* **12,** 7 (July 1969).

26. B. Randell and C. J. Kuehner. "Dynamic Storage Allocation Systems." *Comm. ACM* **11,** 5 (May 1968): 297–306.

27. "RCA SPECTRA 70/46 Reference Manual." 70-46-601, March 1968.

28. A. L. Scherr. "Functional Structure of IBM Virtual Storage Operating Systems—Part II: OS/VS2-2 Concepts and Philosophies." *IBM Systems Journal* **12,** 4 (1973): 382–400.

29. J. L. Smith. "Multiprogramming under a Page on Demand Strategy," *Comm. ACM* **10,** 10 (October 1967).

30. "3300 Computer System Reference Manual." Control Data Corporation, Pub. No. 60157000, 1966.

31. "3300/3500 Computer Systems—Master—Reference Manual." Control Data Corporation, Pub. No. 60213600, April 1969.

32. V. A. Vyssotsky, F. J. Corbató, and R. M. Graham. "Structure of MULTICS Supervisor." *Proc. AFIPS 1965 FJCC* **27,** 1 (1965): 203–212.

第 6 章

データベース管理システム

注　意

この章の目的は，データベース管理システムを概括することではない．そのかわりに，システムの最も重要な2つの側面について集中的に学んでいる．すなわち，

1. データ構造，アクセスおよび更新．
2. データの機密保護．

これらを学ぶために，抽象的なモデルを導入した．抽象化はある程度形式的なものであるけれども，それは，実際に稼動している管理システムから注意深く導き出している．

最初にデータ構造——アクセスと更新——をモデル化し，次にデータの機密保護をモデル化する．モデルを学ぶと，データベース管理システム内の基本概念や流行の考え方を，そこに含まれるシステムの要素を詳細に示すことをしなくても我々はよりよく理解することができる．

モデルはかなり完全にできているので，特定のデータベース管理システムについての事前の知識はここでは要らない．それにも拘わらず，読者は，順次ファイルや索引順序ファイルはどんなものであるかを知っている必要があり，ディスクのような乱アクセス記憶機器の構成や使い方をよく知っていなければならない．

6.1 は じ め に

将来のオペレーティング・システムは主としてデータベース管理システム向きになるという徴候があるので，データベース管理システムは重要である．この徴候は，計算機システムのオンライン的使い方が増大していること，また

非常に低廉で大容量，低速の補助記憶から，非常に高価で，小容量，高速の固定ヘッドの回転型機器に至る範囲のオンライン記憶を提供できる多重レベル二次記憶機器が多様化し大量に導入されていることにあらわれている．したがって，オペレーティング・システムを物理資源の管理者と見なすという伝統的な考え方は狭すぎる．さらに，主記憶，仮想記憶，入出力操作，データチャネル，バッファの観点から物理資源を管理するという問題は，よく理解されている．一方，将来の計算機システムにおけるオンラインのデータベースは巨大なので，オペレーティング・システムはさらに論理資源管理者になる必要がある．論理資源の管理は，データベースにおける構成，機密保護，アクセス，プログラムやデータの記憶などに関係してくる．このために，データベースの構造，保護，使用に対して統一的に対処できる一般化したデータベース管理システムの抽象モデルを提案してきた．

統一的なやり方を促進してきた理由は，たくさんある．

1. 見かけ上多様な構造をもっているデータベース管理システムも，共通の性質をもっている．したがって，よく知られている多数の構造を特殊な場合として含むような一般化したデータ構造を使うことによって，構造の統一的な扱いを提案することが可能である．データ構造とは，二次記憶上でデータをオンラインで蓄積，アクセス，更新するための論理レコードの構成とファイルの構造とを意味している．データベースシステムの中で多数のデータ構造が必要になるのは，異なったデータ構造は異なったデータベース管理の仕事を実現できるという事実に基づいている．たとえば，索引順序ファイル構造は，プログラムのファイルの管理を実現できるが，そこでは，プログラムの文の行番号が索引(キーワード)として使え，プログラムの文はレコードとして処理し，プログラムはファイルとして扱われる．他のデータ構造は，問い合せ，レポート出力，トランザクション向きの操作などができる．しかしながら，商用のデータベース管理システムは，少数のデータ構造を扱うことしかできない．それゆえ，一般的なデータ構造を研究することにより，多くの見かけ上多様な構造を一般化したものの特殊例として見ることができるだけでなく，これらの構造を主として扱っているデータベース管理システムの設計と実現を追及することができる．

2. データベース管理システムにおけるいろいろのアクセスや更新の方法は，また，お互いどこか本質的に似ている．データベース管理システムの中で単に多数のデータ構造を使えるだけでは，必ずしも効率の良いシステ

ムにはならないであろう．これらのデータ構造は二次記憶に対するものだから，構造化したデータに効率良くアクセスしそれを更新する方法も必要である．したがって，データ構造を導入するときにはいつでもアクセスや更新の方法も含まれるということは，別に驚くことではない．一つの良い例が，索引順序に付随しているISAM (IBMのIndexed Sequential Access Method)である．データベース管理システムにおけるデータ構造の多様さは，アクセスや更新の多様な方法を必要とする．データベース管理システムが多数のデータ構造と更新の方法を必要としているので，それによってデータベース管理システムの設計や実現だけでなく，これらの方法の議論をも複雑にしてしまう．これらの方法が本質的に似ているので，特に一般化したデータ構造に対して統一的なアクセスや更新の方法を工夫することが可能である．この方法は一般化した構造に対して働くので，すべての特殊な場合にも働く．更に，アクセスや更新の方法は，個々の特殊な構造に対して少くともそれ用に作った方法と同じ効率であることを示せる．

3. データベース管理システムにおけるアクセス制御やプライバシの保護の機構は，どのデータ構造やアクセスと更新の方法が使用されようと，必要になる．したがって，アクセス制御やプライバシの保護を一般化したシステムの環境で議論するのは，より適切である．データベース管理システムにおける制御や保護は，統合的なデータベースが知的に成長することを強調すれば，ますます必要になる．ユーザは仲間のユーザの経験——過去の失敗，現在の努力，将来の計画——から利益を得られるのはデータを共用することによって行なわれるので，知的に成長するためには，システムはデータを共用するようにさせなければならない．しかしながら，データ共用の基本は，自発的なものである．言い換えれば，データベースの所有者が自分のデータのある部分を公開したり，少数のユーザに使わせたり，誰にも使わせないと希望したときに，それが可能でなければならない．自発的でない情報共用システムとは，システムが一度格納した情報のあらゆる部分が，システムのすべてのユーザで使えるようなものである．そのようなシステムは，もっと制限のついたシステム，たとえば，システムのユーザが同一のプロジェクトに属しているような場合には，利点がある．しかし，種々の訓練を受けたユーザや異なった程度の感度をもつデータのある一般的環境では，データの共用を強制することは，望ましくない．そのよ

うな強制は，システムのユーザを失望させることにもなりかねない．

データの共用においては，専用の情報のプライバシと安全保護の問題が生じてくる．データの共用を自発的にするためには，システムは，ユーザのデータのプライバシの保護とそれへのアクセスの制御をする手段を提供しなければならない．プライバシの侵害から守られていることを一度ユーザが知ったならば，システムを信頼して使うようになるであろう．そして，ユーザは個人用のデータベースを作り，データを蓄積し，それを改良し，システムのアクセス制御のもとで，ユーザの意志に基づき何人かの選んだ人に自分のデータの一部あるいはすべてを使わせたり，あるいは，公開するように徐々になってゆくであろう．

6.2 データ構造

6.2.1 相似性

最近のデータベース管理システムの多様性は，プログラミング言語の初期の頃の発展を思い出させる．言語の開発においては，通常，その言語が設計のねらいとしている応用の種類を識別する試みから始まる．この過程では，応用で使われるデータの型と操作の種類とを特徴づけ分類することを行なう．応用が記号処理とか数値計算とか事務用データ処理とか何であっても，言語の開発において最も重要な事は，便利で効率の良い操作が実現できるようなデータ型とデータ構造とを選択することである．適当なデータ構造がなければ，応用におけるデータをその言語の中で容易に記述することができない．適切なデータの表現がなければ，効率の良い手続きを開発することは難しい．勿論，我々は，数値計算に対する配列，記号処理に対する列，データ処理に対する階層を使うことは良く知っている．データ構造を一度選んでから，次に，構造を処理し操作の列である手続きの要素となる操作を開発する．異なった言語は構文，語彙，規則などに大きな差が見られるけれども，これらの差は，大部分の場合人間的要因と美学的考察とによって生ずる．基本算術演算，論理演算，入出力操作の差ですら，比較的小さい．一つの言語と別の言語との実際の違いは，データ構造と，それらのデータ構造を処理するために提供されている効率の良い操作である．

そのような違いを頭に入れて，プログラマは，たとえば，数値的応用のためにFORTRANにおける配列と数学的関数を，記号処理のためにSNOBOLにおける列とその操作を，また，データ処理のためにCOBOLにおける階層とその予約語とを利用するだろう．しかし，応用がこれらのすべてのデータ構造と

処理および計算の広汎な要求とを含んでいるときに，困難な問題が生ずる．この場合，プログラミング言語が，データ構造の豊富な集合と，それらの構造に対する効率の良い演算子や関数の大規模な集合を持っていることを要請する．PL/I は，これらの機能を提供しようと試みている．応用の複雑さが増すにつれて，非常に多様性のあるデータ構造，演算子，関数を必要とするのは自然である．しかし，計算の資源がそれを扱えなければ，そのような多様性は経済的でない．したがって，PL/I は，大容量の計算機が多くの組織体で手の届く範囲になってきた最近になるまでは，場面に登場してくることはなかった．

似たような発展が，データベース管理の場合にも生じている．データベース管理システムが必要なのは，二次記憶機器上で大規模なデータベースを扱いたいという要求が動機になっている．各データベース管理システムに対し，二次記憶内のデータの構成を表現するデータ構造は，通常，効率的な格納，アクセス，更新に対する視点から選択され開発される．異なったデータ構造は，それぞれ異なったデータベース管理の要求に向いており，少数のデータ構造しかこれらすべての要求に単独で答えられないので，データ構造およびデータベース管理システムが多様化してくる．たとえば，SDC の TDMS は逆ファイルを特別視し，IBM の GIS は索引順序ファイルだけを許し，Xerox の DATRIX は多重リストファイル構造を専ら利用している．しかし，データベースの最近の成長の現象と，広範囲の応用を支援することへの要求が増大していることを考えると，データベース管理システムの有用性は，個々のデータ構造を特別に扱える能力だけにあるのではなく，一方で既存のあるいは新しいデータ構造の豊富な集合を与える能力に，また一方で，これらの構造に対する効率の良いデータベース管理の機能の大規模な集合にもある．この章において，可変構造で高速アクセスのデータベース管理システムのモデルを議論するのは，これらの機能を提供したいと思っているからである．

6.2.2 可　変　構　造

データ構造は，2つの異なった方法，すなわち，レコードの構成とファイルの構造とから考えられる．データ項目の特性と，格納およびアクセスに対する単位としてデータ項目を形づくることを考えるのは，レコードの構成の研究である．一つのレコードは一個以上のデータ項目から構成されるけれども，これは，一時に格納したりアクセスしたりできるデータの最小単位である．このようなレコードの概念をもっていれば，モデルは格納やアクセスの処理を最適化でき，効率の良いデータベース管理機能を提供することができる．

レコードの構成が，レコード内の個々のデータ項目がどのように相互関係をもっているのかを決めるのとちょうど同じように，ファイル構造がファイル内の個々のレコード間の相互関係を決定する．ここでは，広く使用されているファイル構造が特殊な場合として含まれるような一般化したファイル構造を利用する．一般化したファイル構造を理解する鍵は，ディレクトリとファイル構造の概念の基本になっているキーワードとリストの概念である．

6.2.2.1 レコードの構成

定義を与えない2つの術語，すなわち，“属性”の集合 A と“値”の集合 V とを考える．**レコード** R とは，各属性が只一つの値をもっているような直積 $A\times V$ の部分集合である．したがって，R は，(属性，その値)という形式の順序対の集合である．

レコード R に対する**索引**(index)とは，属性-値の対の集合で，それが R の特性を決める．実際上，比較的記述が短かく単純である属性-値の対を選んでおくと，役に立つ．索引におけるこれらの順序対を，**キーワード**という．記号 K または Ki は，キーワードを示すために使われる．

すべてのレコードには，**番地**とよぶ一意的な正の数を割り当てる．これは，与えられた記憶空間内のどこにレコードがあるのかを示している．計算機システムでは，記憶空間は，多種類の記憶機器から構成されているかもしれないが，レコードにはその記憶機器に応じて一意的な番地を割り当てなければならない．

ある場合には，レコード R に，それと同じキーワードを持つ別のレコードの番地をその一つのキーワードとして与えられていることがある．このような番地を，K に関する R の**ポインタ**とか，もっと簡潔には，R の K ポインタと呼ぶ．R が K に付随した番地を含んでいなければ，K に関して R は**空ポインタ**を持っているという．**データベースのレコード**では，レコードのすべてのキーワードにポインタがついている．これ以後，レコードとは，データベースのレコードを意味するものとする．

キーワード K に関するレコードの**リスト** L，あるいはもっと簡潔に K **リスト**とは，次のような条件を満す K を含んだレコードの集合である．すなわち，

a） K ポインタは，異なっている．

b） L 内の空でない K ポインタは，すべて，L 内のレコードのレコード番地を与え，かつ L 内のものだけである．

c） L 内に K を含んだ他のどのレコードからも指されていない一意的な

レコードがある．これを，リストの**先頭**という．

d)　L 内に空の K ポインタを含む一意的なレコードがある．これを，リストの**終端**という．

e)　L 内のレコード番地が $a_n (n > 1)$ であるような任意のレコード R に対し，K ポインタの列

$$(a_1, a_2, \cdots, a_n)$$

が存在する．ただし，

i)　a_1 は，L の先頭の番地である．

ii)　$j = 1, 2, \cdots, n-1$ に対し，番地が a_j であるレコードは，K ポインタ a_{j+1} を含む．

したがって，同一のキーワード K に対し2つの K リストがあるならば，それらは，共通のレコードを持っていない．

たとえば，**図6.1** では，レコード R は6個のキーワードをもち，そのうちの一つ K_1 には，空ポインタが割り当てられている．この場合，空ポインタは，K_1 を含んでいる他のどのレコードも指していないことを知っている．一方，R の K_6 ポインタは，R から R' へ向かい．今度は，R' の K_6 ポインタは R' から R'' へ向かっている．キーワード K_6 で決まるこの有向路は，R'' の K_6 ポインタが空なので，ここで終了する．

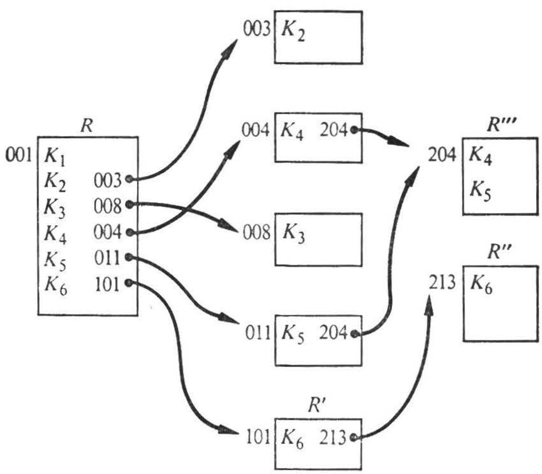


図 6.1　K リスト

この例では，ちょうど6個のリストがあることに注意しよう．

K_1 リストは，単一レコード R を含んでおり，これは，ちょうど一点からなる有向路と見なすことができる．

K_2 リストは，番地 001 と 003 にあるレコードを含んでおり，したがって 001, 003 という(有向)路を作っている．

K_3 リストは，001，008 という路をもつ．

K_4 リストは，001，004，204 という路をもつ．

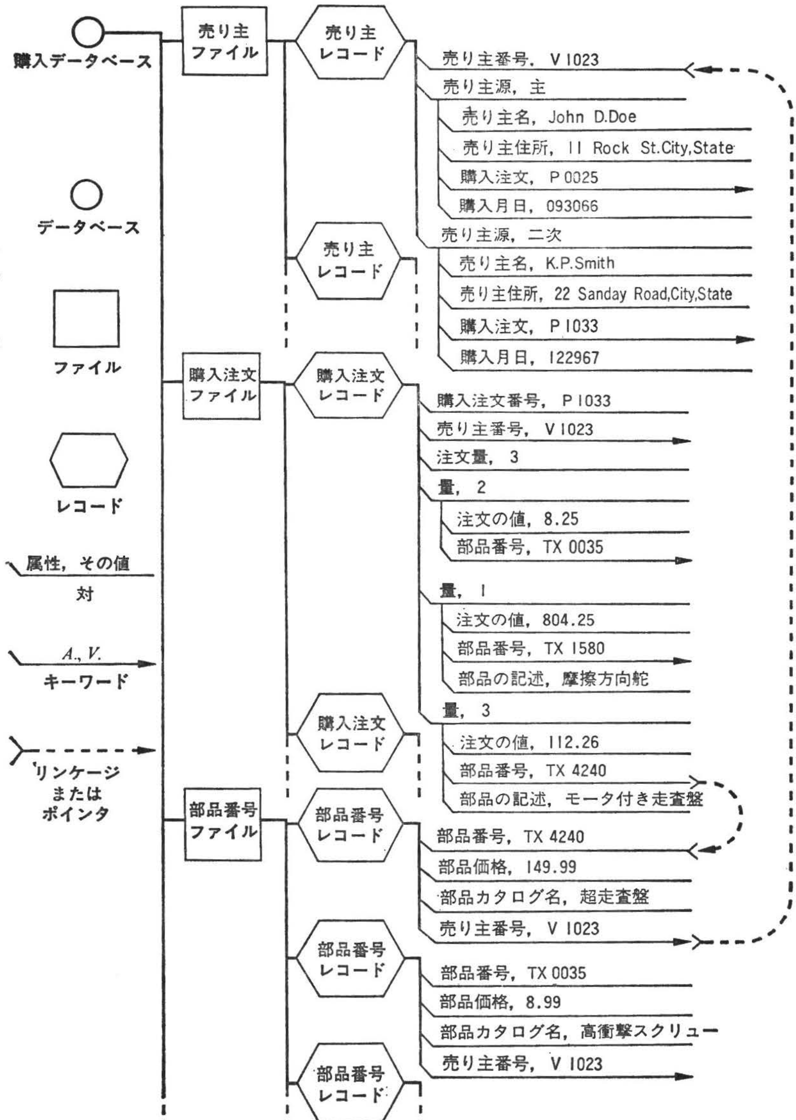
K_5 リストは，001，011，204 という路をもつ．

K_6 リストは，001，101，213 という路をもつ．

レコードの集合 F は，1個以上のレコードを含むすべての K リストが F に含まれているとき，**ファイル**であるという．いくつかのファイルの中で区別するために，すべてのファイルは，**ファイル名**と呼ぶ一意的な名前をつける．

ファイル内のデータ項目の情報の型と構成とを明らかにするために，属性-値の対の特性と他の属性-値の対との関係を指定することができる．ファイルの属性-値の仕様が，ファイルの**レコード構成**(record organization)を決めるという．直観的には，一つのファイルは1個以上のレコードから構成されてもよく，一つのレコードは，1個以上の属性-値の対から組み立てられていてもよい．キーワードとして指定されている属性-値の対は，ポインタをもっている．ファイル，レコード，属性-値の対およびキーワードポインタの関係を，図 6.2 に示す．データベース全体は，四角の節点で示している多くのファイルから構成される丸い節点であらわす．ファイル名は，節点の中にある．各ファイルは，六角形であらわした多くのレコードから作られる．各レコードは，属性-値の対の階層として示してある．属性とその値とは，コンマで区切ってある．キーワードは，矢印のついた下線を施した属性-値の対である．いくつかのポインタは，破線で示し，同一のキーワードのレコード番地を指定している．

ファイル内のすべてのレコードは同一の情報の型とレコードの構成とをもっているので，一つのレコードの属性-値の対は，いつも，同じ情報の型をもち，他のレコードの属性-値の対と同じように相互に関係をもっている．一つのレコードが他のレコードと異なる点は，属性の値と，レコードの構成における選択的な属性-値の対があったり無かったりすることである．たとえば，図 6.2 では，部品番号ファイルのすべての部品番号レコードは，同一の情報の型(属性)と同一の構成(4つの対が同じ順序で同じ段の刻みで関係づけられている)とをもつが，その属性に対して与えられた値は異なる(たとえば，一つのレコードの部品番号は TX 4240 で，もう一つは TX 0035 である)ことに注意しよう．



購入データベース
データベース
ファイル
レコード
属性，その値
対
A., V.
キーワード
リンケージ
または
ポインタ
売り主ファイル
購入注文ファイル
部品番号ファイル
売り主レコード
購入注文レコード
部品番号レコード
売り主番号，V 1023
売り主源，主
売り主名，John D.Doe
売り主住所，11 Rock St.City,State
購入注文，P 0025
購入月日，093066
売り主源，二次
売り主名，K.P.Smith
売り主住所，22 Sanday Road,City,State
購入注文，P 1033
購入月日，122967
購入注文番号，P 1033
売り主番号，V 1023
注文量，3
量，2
注文の値，8.25
部品番号，TX 0035
量，1
注文の値，804.25
部品番号，TX 1580
部品の記述，摩擦方向舵
量，3
注文の値，112.26
部品番号，TX 4240
部品の記述，モータ付き走査盤
部品番号，TX 4240
部品価格，149.99
部品カタログ名，超走査盤
売り主番号，V 1023
部品番号，TX 0035
部品価格，8.99
部品カタログ名，高衝撃スクリュー
売り主番号，V 1023

図 6.2　レコード構成

購入注文ファイルでは，いくつかの注文には説明があり，他のものにはない．

ここで述べているレコードの構成は，COBOL における仕様の機能をすべて完全に含んでいる次のような属性-値の仕様の機能によって特徴づけられる．このレコード構成を，**一般化したレコード構成**(generalized record organization)と言う．

A. 属性-値の対のレベル

すべての対には，レベル番号を与える．すべての対につけたこれらの**レベル番号**(level number)は，階層的なレコード構成を実現する．それゆえ，レコードは木構造と見てよいが，そこでは，木の節点が属性-値の対であり，木の節点の深さが，レベル番号をあらわす．木の根とその対に対しては，レベル番号0を割り当てる．深さ N の節点は，レベル番号 N を割り当てた属性-値の対をもっている．

木の与えられた非終端節点に対し，その節点が根となるような部分木が必ず存在する．その節点に付いている属性-値の対を**上位対**(superior pair)，その部分木に付けられた属性-値の対を，その(上位)対の**従属対**(subordinate pair)とよぶ．明らかに，属性-値の対のレベル番号は，各従属対のレベル番号よりも小さい．特に，レベル番号が上位対のレベル番号よりも一つだけ大きい従属対を，その(上位)対の**直下従属対**(immediate subordinate pair)という．逆に，その上位対を，直下従属対の**直上上位対**(immediate superior pair)という．

図6.2では，同一のレベルにある属性-値の対は，同一の縦線から書くという規則を使っている．もっと詳しくいうと，売り主番号と売り主源を含む対は，同一のレベルにある．売り主名，売り主番地，購入注文，購入月日は，一つのレベルにあるけれども，このレベルは，売り主源を含む対のレベルよりも“下位”にある．したがって，これらの対は，売り主源を含む対の直下従属になっている．対に対する番号は，プログラマが陽に割り当てるか，あるいは，指定無し(default)による．指定無しで割り当てる場合，対に割り当てたレベルの番号は，事前の対に割り当てられたレベルの番号と同じものになる．指定無しの場合，レコードの最初の対に対しては，いつもレベル番号01を割り当てる．レコード内のレベルとか階層から見た対の構成は，対の読み込みや修飾を簡単にするために，副グループ，グループ，部分などとして関係づけられる対を見つけるときに使うことができる．一つのレコードにおける属性-値の対のレベルの例を，**図6.3**に示す．

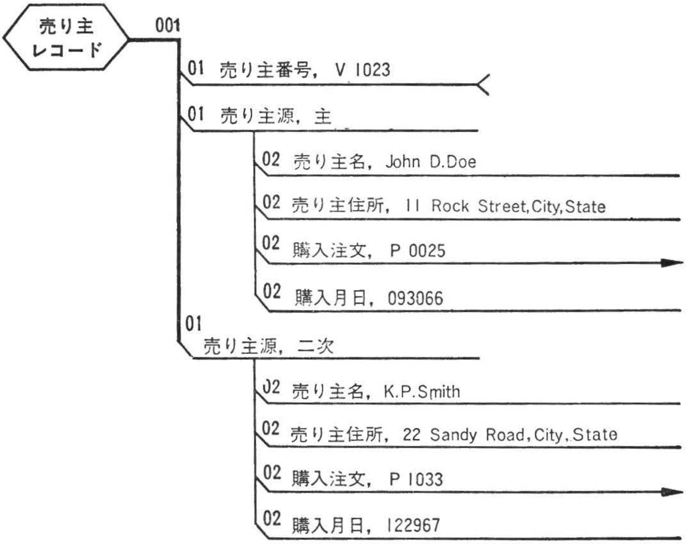


図 6.3 属性-値の対のレベル

B. 属性の出現

図6.2の購入注文を見ると, “部品の記述” という属性が最初 “数量1” の下に現われ, 次に “数量3” の下に現われる. しかし, “数量2” の下には, 部品の記述はない. これは, レコードの中に属性が一様には現れないという例である. さらに, 属性は, ファイルのすべてのレコードに一様には現れないかもしれない. そのような属性の出現を, 属性の**選択的出現**(optional occurrence)という. 一方, 属性 “購入注文番号” は, すべての購入注文レコード内に現れる. 属性のそのような出現を, 属性の**一様(または指令的)出現** [uniform (or mandatory) occurrence] という.

さらに, 属性-値の対の属性の出現が選択的の場合, その直下従属の対の各属性の出現は, 次のように決められよう.

a) その属性が一様で, その直上上位対の選択的属性が出現しているならば, それは出現しなければならない.

b) その属性が一様であっても, その直上上位対の選択的属性が出現していなければ, それは出現できない.

それゆえ，属性-値の対の属性の出現が一様か選択的かということは，その直上上位対の属性の出現が一様か選択的かということに関連してきまる．

C. 属性の繰り返し

レコードの中では，ある属性は1度だけ出現し，別のものは何回も出現してもよい．たとえば，図6.2では，次の属性，すなわち，売り主源，売り主名，売り主住所，購入注文，購入月日は，レコード内に2度現れる．これは，事務の報告書や記録によく見られる現象である．しかしながら，これらの属性を，レコード内で繰り返さねばならない時にいつも指定するのは，かなり面倒である．繰り返し現われる属性を指定するプログラマの労力を減らすために，属性に繰り返し数を割り当てる．レコードに属性があらわれたときには，その属性の**繰り返し数**(repetition number)は，レコードの中でその属性が繰り返される回数を示す．さらに，繰り返す属性は，元のものと同じレベルにあり，元のものよりも下位にあるすべての属性も同様に繰り返される．繰り返し数のないものは指定無しであり，繰り返し数は0である．それゆえ，たとえば“売り主源”の指定においては，それに繰り返し数1を割り当てる必要がある．さらに，売り主名，売り主住所，購入注文，購入月日の指定においては，繰り返し数を指定する必要はない．繰り返し数の無いのは，繰り返しが起らないことを陰に示している．これらの4つの属性は繰り返される属性より下位にあるので，“売り主源”という属性が繰り返されるごとに，それらも(図6.2に示したように)繰り返される．

D. 値の型

属性の値は，英数字，整数，実数，パック型，アンパック型，2進形式などでよい．そのような仕様は，その値を扱うときに，その処理プログラムにとって役に立つ．

E. 値の大きさ

値の大きさとは，その値を構成する文字数である．この数は，固定あるいは可変であろう．その数が固定の場合は，その属性の値は，その属性が現われるすべてのレコードで同じ大きさを占めるであろう．そうでない場合は，値の大きさは，レコードごとに変っている．

F.　キーワードの指示

属性-値の対は，アクセスのためにキーワードとして使っても使わなくてもよい．たとえば，“売り主名”という属性のような短かい対は，検索文の中で使うかもしれないが，“売り主住所”という属性のように長いものは，実際的な理由により使われないであろう．ユーザが以後の検索や問い合せのためにキーワードとして使うことを指示された属性-値の対は，ファイルに対するキーワードディレクトリを構成するために，データベース管理システムで使うことができる（ディレクトリとファイルに関する議論は，以後の節で行なう）．

G.　リンケージとポインタ

別々のファイルに属しているレコードが，同一の属性と値とをもっているために，お互に“関係し合ってる”かもしれない．たとえば，売り主からの部品の購入は，次の3つのレコードに記録される．すなわち，購入の費用と購入を行なった売り主とを示している購入注文レコード，売り主源と購入月日を記してある売り主レコード，および，販売価格，名前，その供給者を示している部品レコードである．これら3つのレコードは，異なった3つのファイル，すなわち，購入注文レコードは購入注文ファイルへ，売り主レコードは売り主ファイルへ，部品番号レコードは部品番号ファイルへ属している．しかし，これはすべてこの購入と関連している．図6.2に破線で示しているように，購入注文レコード内の“数量3”の下にある“部品番号 TX 4240”の対は，部品番号 TX 4240 の部品番号レコードを参照している．同じ部品番号レコード内で，“売り主番号 V 1023”の対は，売り主番号 V 1023 である売り主レコードを参照している．これら3つのレコードは異なった3つのファイルにあるが，在庫価格が 112.26 ドルで販売価格 149.99 ドルである部品 TX 4240 は売り主 K. P. Smith 氏から購入したので，これらは“関係をもっている”ことに注意しよう．これらのレコード間の相互参照を能率よく行なって，ユーザの応用プログラムが関連のある情報に対して破線を介して“追跡”できるようにするために，これらのフィールドにリンケージがなければならない．たとえば，修飾したファイル名，キーワードの論理式や属性を，他のファイル内のレコードの属性-値の対を参照するための論理的なポインタとして使用できる．一方，レコードの一つの属性-値の対（すなわち，キーワード）を同一のファイルの他のレコードの同じ属性-値の対と関連づけるためには，修飾したファイル名を使う必要はない．前に定義したように，レコードの番地をポインタとして使えば充分である．しかし，リンケージやポインタの選び方の規則は，ここで論ずべき

題目ではない．ここで提案したいのは，ユーザがそのようなオプションを指定できるようにすることである．

今までの議論を要約すると，属性-値の対は，次の8つのパラメータで特徴づけられる．

1. 属性 A;
2. その値 V;
3. そのレベル番号 L;
4. A の出現が一様($O=1$)あるいは選択的($O=0$ または空白)であるかを示す O;
5. V の型 T, たとえば，$T=I$(整数), $T=A$(英数字), $T=D$(10進), $T=$ (値なし)など;
6. V の大きさ S;
7. キーワード($k=1$)または非キーワード($k=0$)を示す指示子 k;
8. ポインタまたはリンケージを必要とする($P=1$)か，あるいは，必要としない($P=0$ または空白)かを示す P.

従って，値を除いてしまうと，レコードの構成におけるすべての属性-値の対に対して，次に示すようにその情報の型と他のレコードとの関係を陽に指定することができる．

$$(A, L, O, T, S, k, P).$$

属性 A に繰り返し数 R が付けられている場合は，A は R 回繰り返されることになる．その繰り返しごとに，新しい対ができる．さらに，L よりも大きなレベル番号をもつすべての対で，その属性-値の対に従属しているものも，繰り返される．そのような繰り返しも，新しい対の集合を作る．この新しい対は，同一の値 V をもつかもしれないし，そうでないかもしれない．情報の型，および，属性が A でその繰り返し数が R であるような属性-値の対の集合の中の対との関係を特徴づけるために，上記の組を，次のように変更する．

$$(A, L, R, O, T, S, k, P).$$

6.2.2.1.1 レコード型紙の概念

レコードは同一の型(属性)の情報を含んでいるので，一緒に集まるとファイルを構成する．たとえば，図6.2を参照すると，売り主ファイル内のすべてのレコードは，属性“売り主源”をもっていると思われる．それゆえ，属性の全

体の数は大きくても，ファイル中の各々の新しいレコードでは，異なった属性の数は限られていると予想しても，理にかなっている．同一のファイル内のレコードがこのような特性をもっているので，レコード自身から属性を取り除いてしまい，一意的な属性をそのファイルに付随したどこか中央の場所に格納してしまえば，かなりのファイルの空間を節約することができる．この中央の場所を，**レコード型紙**(record template)あるいは簡単に**型紙**と呼ぶ．一つのファイルに集められているレコードは，同一の型(属性)の情報をもっているだけでなく，同一の一般化した構成をしていることに注意するのは，重要である．したがって，型紙は，一意的な属性をもつだけでなく，属性-値の対の間にある関係に関する情報ももっている．これらの関係は，ファイルの一般化したレコード構成を決めるので，この情報を，ファイルの**全域的レコード構造仕様**(global record structure specification)という．

個々のレコード内に格納すべき他の情報として，属性の値のようないくつかの構造に関係ない情報に加えて，与えられたレコード構成に対するある全域的構造の機能があるかないかを示す情報もある．レコードに局所的であるこのような情報は，そのレコードの**レコード制御情報**(record control information)という．

これを考慮して，すべての一意的属性に対する8つ組を型紙に置く．各々のユーザレコードに対しては，そのレコード制御情報を格納する**レコード制御ブロック**(record control block)を作成する．したがって，記憶では，一つのレコードはその属性のすべての値とレコード制御ブロックから構成される．ユーザが認識するレコードと記憶内にあるレコードとを区別するために，ユーザのレコードを**論理レコード** (logical record)，その構成を**論理レコード構成**という．それに対し，値とレコード制御ブロックから構成され格納されているレコードを**物理**あるいは**システムレコード**，その構成を**物理レコード構成**という．すなわち，論理レコードはユーザとそのプログラムから見えるものであるのに対し，システムレコードはシステムが処理するものである．

もっと精密にいうと，物理レコード構成は，レコード型紙の概念に基づいていることを学んだ．

レコード制御ブロックの各エントリは，属性-値の対に対応し，

(制御番号，値の大きさ)

の形式をしている．ここで，制御番号は正の整数である．属性-値の対に対して一意的な正の整数を制御番号として割り当てることは，あらかじめ決められ

たアルゴリズムに従ってレコード入力プログラムによって行なわれ，ユーザの関わることではない[†].

レコードを入力する前に，ユーザはレコードの構成の仕様を与える．その仕様により，型紙内の8つ組の集合が決まる．ユーザのレコードが入ってくると，レコード入力プログラムは入ってくるレコードをそれぞれ型紙に対してチェックする．レコードが型紙内の全域的構造仕様に合致すれば，レコード入力プログラムはそのレコードを処理し，その制御番号を計算し，レコード制御ブロックを構築し，その記憶内のイメージを生成する．ユーザに関する限り，そのレコードの記憶のイメージには属性は入っていず，すべてのレコードに対する一意的な属性は型紙に置かれていることは関知していない．自分のレコードは関連のある対の集合と考え，各対は属性とその値とから構成されていると考えている．

6.2.2.1.2 レコード構成の型

今までの議論を読者にもっと明確に理解させるために，図6.2の図を使って，事例の研究をしよう．

事例 1: UF 型

売り主ファイルにおけるレコードに対するレコード構造仕様は，次のようなものであろう．

(*A* ,	*L*,	*R*,	*O*,	*T*,	*S*,	*k*,	*P*)
(売り主レコード,	00,	,	1,	,	,	,	)
(売り主番号,	01,	,	1,	*A*,	5,	1,	1)
(売り主源,	01,	1,	1,	*A*,	9,	1,	)
(売り主名,	02,	,	1,	*A*,	20,	1,	)
(売り主住所,	02,	,	1,	*A*,	50,	,	)
(購入注文,	02,	,	1,	*A*,	5,	1,	1)
(購入月日,	02,	,	1,	*I*,	6,	1,	)

この仕様によれば，各々の売り主レコードは，同じ数の属性をもっている．さらに，各々の値は，固定の大きさである．それゆえ，この仕様によって生成されるレコードは，同一の長さをもっている．属性の出現は**一様**(uniform)で

† この議論では，アルゴリズムは，レコードの対を木の節点として扱う(根に対して，空の節点がくることもあり得る)．各節点には，木の根から始めて右側の部分木をたどり，次に左側の部分木をたどって正整数を割り当てる．木をたどるアルゴリズムに関しては，[16] を参照するとよい．

あり，その値の大きさは**固定**(fixed)であるので，このレコードの構成を**一様固定型**(**UF型**)という．UF型のレコードでは，個々のレコードに対してレコード制御情報を持たせる必要はない．それゆえ，これらのレコードの記憶イメージは，レコード制御情報をもってはいない．レコード入力プログラムによって生成される制御番号を，次に示すように各々の属性-値の対の仕様の左側に並べる．

	(*A*	*L*,	*R*,	*O*,	*T*,	*S*,	*k*,	*P*)
0	(売り主レコード,	00,	,	1,	,	,	,	)
1	(売り主番号,	01,	,	1,	*A*,	5,	1,	1)
2, 7	(売り主源,	01,	1,	1,	*A*,	9,	1,	)
3, 8	(売り主名,	02,	,	1,	*A*,	20,	1,	)
4, 9	(売り主住所,	02,	,	1,	*A*,	50,	,	)
5, 10	(購入注文,	02,	,	1,	*A*,	5,	1,	1)
6, 11	(購入月日,	02,	,	1,	*I*,	6,	1,	)

3番目の仕様は，繰り返し数1をもっているので，2つの制御番号を割り当てられている．2つの制御番号をもつ他の対は，レベル番号がそれより高いものに対して従属している．制御番号から，すべての売り主レコードは，ちょうど11個の属性-値の対をもつことがわかる．さらに，売り主番号，売り主源，売り主名，購入注文および購入月日が属性である対は，キーワードとして指定でき，またレコードは同一の売り主番号や購入注文をもっている他のファイル内のレコードと結合されていることがわかる．

事例2: OF型およびOV型

購入注文ファイルの中のレコードに対するレコード仕様は，次のようなものであろう．

	(*A*	*L*,	*O*,	*T*,	*S*,	*k*,	*P*)
0	(購入注文レコード,	00,	1,	,	,	,	)
1	(購入注文番号,	01,	1,	*A*,	5,	1,	1)
2	(売り主番号,	01,	1,	*A*,	5,	1,	1)
3	(注文量,	01,	1,	*I*,	1,	,	)
4, 8, 12	(量,	01,	1,	*I*,	1,	,	)
5, 9, 13	(注文の値,	02,	1,	*D*,	6,	,	)
6, 10, 14	(部品番号,	02,	1,	*A*,	6,	6,	1)
7, 11, 15	(部品の記述,	02,	0,	*A*,	,	,	)

最後の仕様における出現の指示子は0に設定してあるが，これは，“部品の記述”という属性がレコード内に一様に現れることはないことを意味している．これに対応して，局所的なレコード制御情報を個々のレコード内に含ませるようにしている．図6.2の最初の購入注文レコードに対して，レコード入力プログラムは，レコード制御ブロックの次のようなエントリを作成する．そこでは，各エントリは，次の形式をしている．

(制御番号，値の大きさ)
(0，)
(1，)
⋮
(6，)
(8，)
⋮
(10，)
(11，5)
(12，)
(13，)
(14，)
(15，8)

(7)が無いのは，この選択的な対に対する属性が，このレコードに対しては生成されなかったことを示している．(0，)，(1，)や他のところで第二要素が無いのは，制御番号0，1や他のものをもつ対の値の大きさは，型紙の中の全域的レコード仕様によって決められることを示している．レコード制御エントリが必ずしもすべて第二要素をもっているとは限らないとき，レコード仕様は，**選択的固定型(OF 型)**という．この場合，個々のレコードにおいて属性が**選択的**(optional)に現われてもよい．しかしながら，値の大きさは**固定**(fixed)である．ここで学んでいる事例では，(11，5)や(15，8)のようなレコード制御情報をもち，個々のレコードが属性の値の大きさを制御できるので，OF 型ではない．この場合，制御番号11をもつ“部品の記述”という属性は，5文字の値をもつ．それに対し，制御番号15をもつ同一の属性は，8文字の値をもつ．少くとも一つのレコード制御エントリが第二要素をもつならば，そのレコード仕様を，**選択的可変型(OV 型)**とよぶ．上記の例は，OV 型のレコード仕様である．

事例3: UV型

最後に，部品番号ファイルのレコードに対するレコード構成の仕様を考えてみよう．各々の部品番号レコードでは，4つの属性が一様に現われることに注意しよう．一つの属性は値の大きさが**可変**(variable)であり，ほかの属性は固定の大きさの値をとる．次に示すのは，**一様可変型**(UV型)のレコード仕様の典型的な例である．

0	(部品番号レコード,	00,	,	1,	,	,	,	)
1	(部品番号,	01,	,	1,	A,	6,	1,	1)
2	(部品価格,	01,	,	1,	D,	6,	,	)
3	(部品カタログ名,	01,	,	0,	A,	,	,	)
4	(売り主番号,	01,	,	1,	A,	5,	1,	1)

各レコードのレコード制御ブロックは，可変な値の大きさを制御するための情報を提供しなければならない．第一のレコードに対しては，

(0,)
(1,)
(2,)
(3, 4)
(4,)

を与える．第二のレコードに対しては，

(0,)
(1,)
(2,)
(3, 8)
(4,)

を与える．

結論として，実際の状況では，レコードの構成は，いろいろの型，レベル，および繰り返しの型をもっていてよいことを強調したい．この研究の助けを借りて，このような多様性を実現できる総括的なスキーマを開発してきた．

6.2.2.1.3　再構成と多重構成

再構成とは，既存の構成の中でレコードの情報の型と全域的な構造を変えて新しい構成を作ることを意味する．ファイルのレコードの情報の型と全域的な

構造の情報は単一の型紙に集められているので，ファイルのレコードの再構成は，型紙の変更につながる．もっと詳しくいうと，型紙の中の８つ組の変更をする必要がある．大きなファイルに対しては，レコードの処理時間に関して，非常に節約ができる．個々の論理レコードがその記憶イメージ（すなわち，値とレコード制御ブロック）を処理しなくても再構成できるので，処理は型紙だけに制限しておく．たとえば，図6.3に示した既存の売り主レコードは，**図6.4**に示すように新しい応用とユーザの要求とを反映したものに再構成してもよい．そのような再構成の結果，上述の事例１に示した型紙を下に示すように変更するだけでよい．

(A	L,	R,	O,	T,	S,	k,	P)
(販売者レコード,	00,	,	1,	,	,	,	)
(販売者識別名,	01,	,	1,	A,	5,	1,	1)
(販売者ファイル,	01,	1,	1,	A,	9,	1,	)

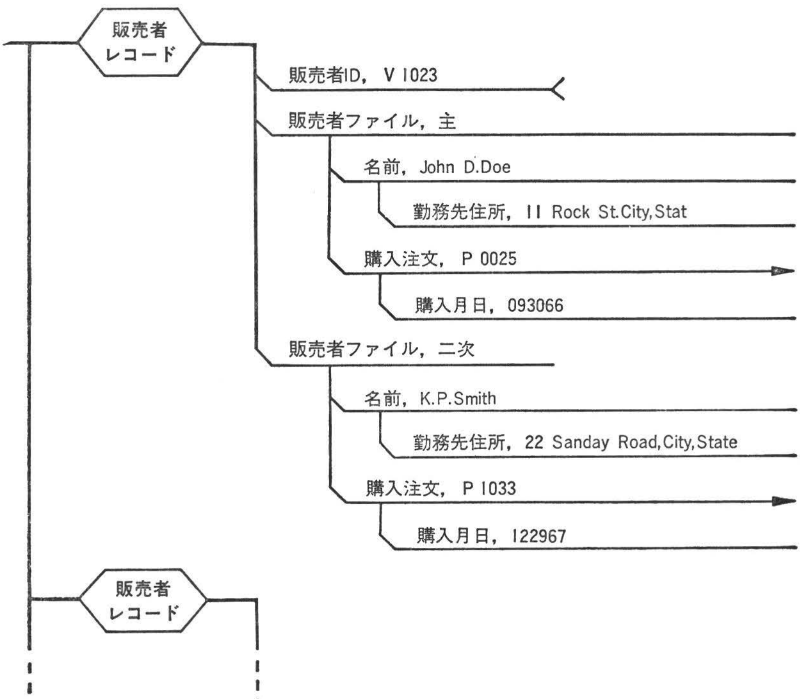


図 6.4 再構成したレコード

(名前,	02,	,	1,	A,	20,	1,	)
(勤務先住所,	03,	,	1,	A,	50,	,	)
(購入注文,	02,	,	1,	A,	5,	1,	1)
(購入月日,	63,	,	1,	I,	6,	1,	)

この再構成においては，元からあるいくつかの属性と，勤務先住所と購入月日とを含んでいる属性-値の対のレベル番号に影響が及んでいたことに注目しよう．

この議論では，レコードに2つの型紙――一つは図6.2に示した元のレコード構成でもう一つは図6.4に示すような新しい構成――が付随していることに注意しなければならない．言い換えれば，同一のレコードの集合に対して，2つのレコードの構成がある．これを考慮して，同一のファイルに対して多重の型紙を保持するようなデータベース管理システムを開発することができる．各々の型紙はファイルのレコード構成を決めるので，多重の型紙をもつファイルは，多重のレコード構成を使えることになる．

要約すると，単純な再構成を説明するために，意識的に初歩的な例を使ってきた．複雑な階層をもったレコードの精巧な再構成のためには，型紙の設計，制御番号の割り当て方のアルゴリズムの選択，および，レコード制御ブロックの構築に対して慎重な考慮が必要である．

6.2.2.1.4　結　論

レコードの型紙を実現するためにはさらにアクセスや処理が必要になるけれども，その利点は十分そのオーバーヘッドを補なって余りある．もっと詳しく言うと，レコードの型紙を使うと，次のことができる．

1. 個々の論理レコードから属性を取り除いて型紙の中に別々の属性を置くことにより，記憶空間を節約できる．
2. 個々のレコードを処理せず型紙だけを修飾することにより，ファイル全体の再構成ができる．
3. 新しい型紙を導入することにより，同一のファイルに対しレコード構成を追加することができる．
4. 応用プログラムをレコードの物理的配置とは独立に書くことができる．
5. 個々のデータ項目やレコードに対してアクセスの制御を強制することを実施できる．

すべてのデータの定義とレコードの構造的な記述とを型紙に集中させるという努力は，いくつかの遠大な効果を目指している．

1. レコードを処理するためには，ユーザ(またはユーザプログラム)は，そのプログラムにおいてレコードのデータ形式やその記述を与えなくてもよい．COBOL の場合，これによって，プログラムのデータディビジョンの大部分が除去できる．
2. データの定義と構造記述の情報を中央の型紙の中にもっているので，異なったユーザ(あるいはユーザプログラム)が，レコードのデータ形式と構成を同じように理解し解釈し，従って，情報の誤った解釈を防ぐことができよう．
3. レコードの所有者は，自分のレコードの別のユーザに対して，元の型紙から導いた新しいレコードの型紙を割り当ててもよい．新しいレコードの型紙は，レコード構成の一部に対してだけ十分なデータ定義と構造記述の情報をもっているだけでよい．その結果，ユーザは，レコード構成の他の部分の仕様がその型紙に指定されていないので，レコードのその部分を使うことはできない． そのような型紙を，**導出型紙**(derived template)という． 導出型紙を使えば，データの保全性と保護とを促進できるであろう．**データ保全性**(data integrity)とは，データが誤ったアクセスによって変更されたり汚されたりしないということである．導出型紙は保全性や保護のための番人となっているレコードの部分に関する情報を何ももっていないので，ユーザ (またはユーザプログラム) は，レコードのその部分にアクセスすることは絶対できない．

6.2.2.2 ファイル構造

前節のリストやキーワードの定義を見ると，同一のキーワードに対する2つのリストが，そのキーワードを含んでいるレコードを共用することはない．さらに，一つのレコードが2つの異なったキーワードにより特徴づけられているとき，そのレコードは2つのリストに属するであろう．直観的には，与えられたキーワードにおいて，システムがレコードに対して一つのリストを介してアクセスできる場合，そのレコードに対し別のリストを介してアクセスする必要がないことを前者は意味している．後者は，多重キーワードに対して，システムが，それぞれのリストを介してどれがこれらのキーワードによって特徴づけられているのかを決めることを意味している．たとえば，**図 6.5** に示したレコード R は，データベース管理システムの中には存在できない．キーワード K_3

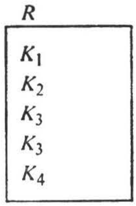


図 6.5　許されないレコード

は R の中に一回以上あらわれるので，R は，K_3 のリストの定義から考えてレコードではない．実際，情報(すなわち，キーワード)がレコードの中に冗長に格納されてよい理由はない．

また，キーワードにより，ユーザは一つのレコードを別のものと関連づけることができ，リストにより関連したレコードへのアクセスの路を確立することができる．レコードのリストを利用するために，ファイルに対するディレクトリと記憶セルの概念を導入する．

6.2.2.2.1　セル，ディレクトリおよび一般化したファイル構造

ちょうど m 個の異なったキーワードをもつレコードのファイルを F としよう．各々のキーワード K に対し，K をもっているレコードの数を n とし，F 内の K リストの数を h とする．さらに，j 番目の K リストの先頭のレコードの番地を a_j としよう．先頭の番地が a_j の K リストにあるレコードの数を p_j とする．定義により，キーワード K_i をもつレコードの数 n は，次のもので与えられる．

$$n = \sum_{j=1}^{h} p_j. \tag{1}$$

$|F|$ は，“ファイル空間”における異なり得る番地の数をあらわすとする．それゆえ，各レコードの番地 a に対し，次式が成り立つ．

$$0 \leq a_{ij} \leq |F| - 1.$$

先頭の番地の集合を考え，$0 < s \leq |F|$ なる固定の正の整数 s に対して次のように定義される関係を E としよう．

2つの先頭番地 a と a' とが関連していること，すなわち，aEa' が成り立つための必要十分条件は，

$$a = es + r$$

かつ

$$a' = es + r'$$

が成り立つことである．ここで，

$$0 \leq r < s \text{ かつ } 0 \leq r' < s.$$

a, a' は正の整数なので，r, r' および e の値は，非負整数である．さらに，E は同値関係† である．それゆえ，a の集合は，**二次記憶セル**あるいはもっと簡潔に**セル**と呼ぶ E による同値類に分割することができる．セルの数は，e の最大値よりも一つ大きい．整数 s は**セルの大きさ**とよばれ，セル内で得ることのできる異なった番地の数となる．いつでも，与えられた番地 a に対応するセルを，次の比によって計算できる．

$$C = \left[\frac{a}{s}\right], \text{ ここで，かっこは，比の整数部を示す.}$$

各セルに対して，**セル番地** (cell address) とよぶ一意的な正整数を割り当てているが，それは，次の式により計算する．

$$c = s \cdot C, \text{ ここで，} c \text{ はセル } C \text{ に対応するセル番地.}$$

セルという概念は，効率の良いレコードの記憶とアクセスの戦術を究明するために導入された．セルというような抽象的で中立的な術語を使うのは，意図がある．セルの現実の世界における解釈は，特定の装備(たとえば，オンライン記憶として装備されているディスクモジュールの数)，アクセス戦術の精密化(たとえば，ディスクモジュールしたがってシリンダを並行的にアクセスするか，あるいは，ディスクトラックを並行的にだけアクセスするか)，および，選択したファイル構造に依存して決まる．したがって，セルは，ある装備ではディスクモジュールであり，別の装備ではディスクシリンダであると考えてよい．したがって，セル番地およびレコード番地の解釈も，同じようにして決められるであろう．

便宜上，キーワード K の先頭番地 a の各々に対して，次の3つ組を導入する．

$$(c, (a, p)).$$

---

† **訳注** 関係 R が (i) xRx (反射律)， (ii) $xRy \Longrightarrow yRx$ (対称律)，および (iii) $xRy, \ yRz \Longrightarrow xRz$ (推移律) を満たすとき，R は同値関係であるという．

ここで，c は a が属し，K リストの p 個のレコードがあるセルの番地である．**与えられた K リストに対し，K リスト内のすべてのレコードは同一のセル内に存在することが必要である．** しかしながら，一つのセルの中に同じキーワードに対して複数個の K リストがあり得る．

この記法を使って，F の**ディレクトリ**を次のような系列として定義する．

$$\{K, n, h; (c_1, (a_1, p_1)), (c_2, (a_2, p_2)), \cdots, (c_h, (a_h, p_h))\}. \qquad (2)$$

ここで，m 個の異なったキーワード毎に，一つの系列がある．

(1)式を使えば，上記の式は，次のように簡単化できる．

$$\{K, h; (c_1, (a_1, p_1)), (c_2, (a_2, p_2)), \cdots, (c_h, (a_h, p_h))\}. \qquad (3)$$

以後，与えられた K に対し，$(c_j, (a_j, p_j))$ のエントリの集合を $D(K)$ と呼ぼう．こうすると，上記の式は，$\{K, h; D(K)\}$ となる．

添字 i を異なったキーワードを識別するために使うと，式(3)は，また，次の様に書ける．

$i = 1, 2, \cdots, m$ に対し，

$$\{K_i, h_i; (c_{i_1}, (a_{i_1}, p_{i_1})), (c_{i_2}, (a_{i_2}, p_{i_2})), \cdots, (c_{ih_i}, (a_{ih_i}, p_{ih_i}))\}$$

および

$$D(K_i) = \{(c_{i_1}, (a_{i_1}, p_{i_1})), (c_{i_2}, (a_{i_2}, p_{i_2})), \cdots, (c_{ih_i}, (a_{ih_i}, p_{ih_i}))\}. \qquad (4)$$

簡略化すれば，$\{K_i, h_i; D(K_i)\}$ とも書く．

一般化したファイル構造(generalized file structure)とは，ディレクトリをもったファイルである．次節以下では，多くの特殊なファイル構造が，一般化したファイル構造から導き出せることを示していこうと思う．

6.2.2.2.2　索引順序および索引乱ファイル構造

索引順序ファイル(indexed sequential file)は，各キーワード K に対し $n = h = s = 1$ となるような一般化したファイル構造である．さらに，任意のキーワード K_i と K_j とに対し，K_i が辞書順序的に K_j の前にあるとき，$a_{i_1} < a_{j_1}$ が成り立つ．それゆえ，そのようなファイルは各レコードごとに唯一つのキーワードしかなく，かつ，ファイル内のレコードはキーワードの辞書式順序に基づいて順次的に格納されている(図 6.6 参照)．

a_{i_1} が整列していなければ，**索引乱ファイル**(indexed random file)という．

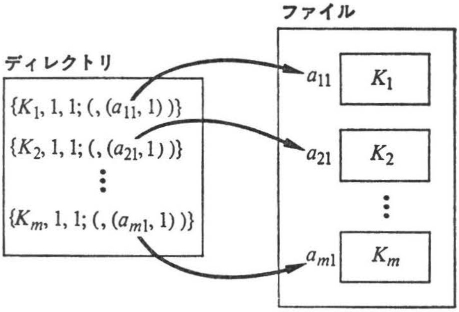


図 6.6　索引順序ファイル

A. 索引順序および索引乱ファイル構造の利用

索引順序ファイルのキーワードの辞書式順序は，レコードの順番を反映しているので，索引順序ファイル構造はプログラムの準備やデバッグをするための（原始）プログラムファイルの維持のために広く使われている．たとえば，FORTRAN のプログラムファイルを，FORTRAN のプログラムの文の行番号をキーワードとし，プログラムの文をレコードとし，プログラム名をファイル名として索引順序的に構成することができる．この応用では，行番号はそれをもっている個々のプログラムの文にアクセスするために使われるだけでなく，プログラムの文の列を維持するためにも利用される．ユーザは一つの行番号からもう一つの（高い）行番号までにあるすべてのプログラムの文を自分のファイルから検索することをデータベース管理システムに要求することによって，プログラムの文に行番号を順番につけたリストを得ることができる．

新しい文を挿入する場合は，ユーザは新しい文にそれぞれ新しい行番号を割り当てなければならない．したがって，新しい行番号，たとえば K_j は，既存の2つの行番号 K_i と K_k との間にあり，ユーザは，新しいプログラムの文を K_i と K_k と番号をつけた文の間に挿入したいことを示している．ここで，もう一度ユーザがそのプログラムの文の新しいリスティングを要求すると，キーワードの英数字の順序に従った系列通りに文がリストされる．すなわち，K_i という番号のついた文は，K_j の番号のついた文よりも前にリストされ，また，K_j の番号の文は，K_k の番号のついた文よりも前にリストされる．プログラムの文の列は行番号（すなわち，キーワード）によって保たれているけれども，レコードの番地（すなわち，a_{i_1}）は $a_{11} < a_{21} < \cdots < a_{m1}$ という条件をもはや満たしてはいない．これは，既存の文（レコード）の間には余地がないので，新し

く挿入された文(レコード)が，ファイルの“終端”に格納されてしまうからである．言い換えれば，初めの索引順序ファイルは，徐々に索引乱ファイルに変化してゆく．それにも拘らず，ユーザは，かなりの量の更新の後では，自分の索引順序ファイルのレコードがかなりランダムに格納されているかもしれないことには，気がつかない．索引乱ファイルのレコードを順次にリストすることは，やはり可能である．しかし，外見上の順列を保つためには，記憶機器のアクセス機構を何回も再位置づけすることが必要になる．

B.　索引順序ファイル構造の実現

索引順序ファイルのディレクトリの実現は，6.2.2.2.2節に述べてあり，また，前章の図5.18に示してある．それを，ここで繰り返すことはしない．実際，構造の実現を以下の節で議論するつもりはない．これらの構造は，すべて，現実のシステムで実現されてきた．紙面の都合上，それらの物理的な実現よりも，それらの論理的な性質について集中的に論じたい．

6.2.2.2.3　逆ファイル構造

逆ファイル(inverted file)とは，すべてのリストが唯一つのレコードをもち，すべてのキーワード K_i に対して，そのセルの番地 c_{i_1}, c_{i_2}, $\cdots$, c_{ih_i} が異なっている，すなわち，$n_i=h_i$ かつ $s=1$ となるような一般化したファイル構造である．これは，すべてのレコードが，セルの全空間を使用していようといまいと，自分自身のセル内に入っていることを意味している．簡単のために，図6.7に示した逆ファイルのセルの番地は示さなかった．

A.　ファイルを逆転するという概念

ファイル処理の初期の頃は，ファイル F は，一意的な番地を割り当てられたレコードの集合と単に考えられていた．一意的な番地を集めて，ディレクトリと呼ぶ集合を作った．ある共通なキーワードをもつレコードの集合 E は一般に F よりもずっと小さいが，それを見つけ出すために，ディレクトリ内のすべての番地を使って，対応するレコードを持ってこようとした．この種のファイル処理に含まれる一般的なステップでは，レコードの番地を得て，次に，そのような番地が与えられているレコードのキーワードを調べる．これらのステップを，ステップの順序を示す矢印を使って，次のように示す．

F 内のすべてのレコード R_i の番地を得る．

↓　　(1)

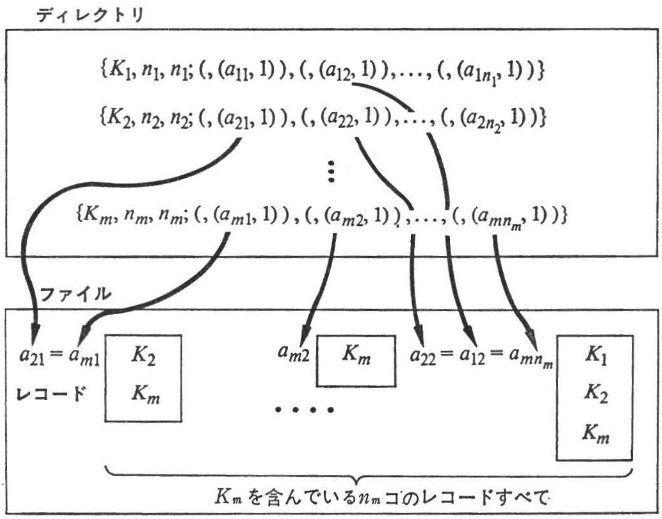


図 6.7 逆ファイル

与えられたキーワードに対し R_i を捜す.

E を得るために F を処理する労力と時間の量と, F 内のすべてのレコードを処理する労力と時間の量との比を r とする. ステップ(1)は, $r=1$ であり, これは, かなり不経済なファイル処理となる.

逆ファイルは上記のファイルと違って, ディレクトリにキーワードの写しをもち, そのキーワードをもつすべてのレコードの番地を各キーワードに割り当てておく. 共通のキーワード K をもつようなレコードの集合 E を見つけるためには, ディレクトリ内の K に付随してある番地をさがすだけでよい.

このステップは,

キーワード K を与える.
↓ (2)
K を含むレコード R_i の番地を得る.

(1)と(2)とを比較すると, (1)ではレコードからキーワードへ行くのに対し, (2)ではキーワードからレコードへ行くので, (2)は(1)から"逆転された"という印象を受ける.

この場合, 比 r は小さくなる. しかしながら, 逆ファイルのディレクトリは, 各キーワードに対してレコードの番地の列が与えられているので, 通常大

きくなる．また，一つのレコードの番地は，異なった列の中で何回も現われてくる(たとえば，図6.7に示したディレクトリでは番地 a_{22} は K_1, K_2 および K_m のエントリとして3回現われる)．それゆえ，ディレクトリの処理のためには，多くの努力が費やされる．

B. 逆ファイル構造の利用

逆ファイル構造には，いくつかの有益な機能がある．

1. 一つのレコードに，一つの以上のキーワードを付けることができる．この機能は，索引順序および索引乱ファイル構造ではないものである．たとえば，レコードが給与支払いのレコードであれば，それは，給与受取者の名前，給与および役職名がキーワードとして必要になる．
2. ユーザは，ファイル内のレコードに関して，レコードを検索してこなくても，統計的な情報を収集することができる．ディレクトリだけを参照することにより，たとえば，データベース管理システムはユーザによって与えられる次のような質問に答えることができる．
   “給与が15,000ドルで，課長職の給与受取者は何人いるか？”

この質問は，ファイルにアクセスしなくても，ディレクトリを処理することによって答えることができる．(給与，15,000ドル)および(役職名，課長)はキーワードなので，これらは，ディレクトリの中に2つのエントリとしてあらわれる．各々のエントリに対して，レコードの番地の列が与えられている．したがって，両方の列に現われる番地を見つけて数えることにより，給与受取者の数が得られる．

一つのレコードに対して多重のキーワードを使うと，レコードをさらに詳細に特徴づけすることができる．レコードを特徴づける各キーワードに対して，レコードの番地をすべて含ませておけば，ユーザはファイルについて問い合わせることができる．これらの機能は，明らかに，索引順序および索引乱ファイルに比べてかなり大量の(ディレクトリの)記憶域と処理費用および労力とを必要とする．

6.2.2.2.4　多重リストファイル構造

多重リストファイル(multilist file)は，キーワード毎にちょうど一つずつリストがあり，セルの大きさがファイルの空間と等しくなること，すなわち，$h_i = 1$ かつ $s = |F|$ が成り立つような一般化したファイル構造である．$s = |F|$

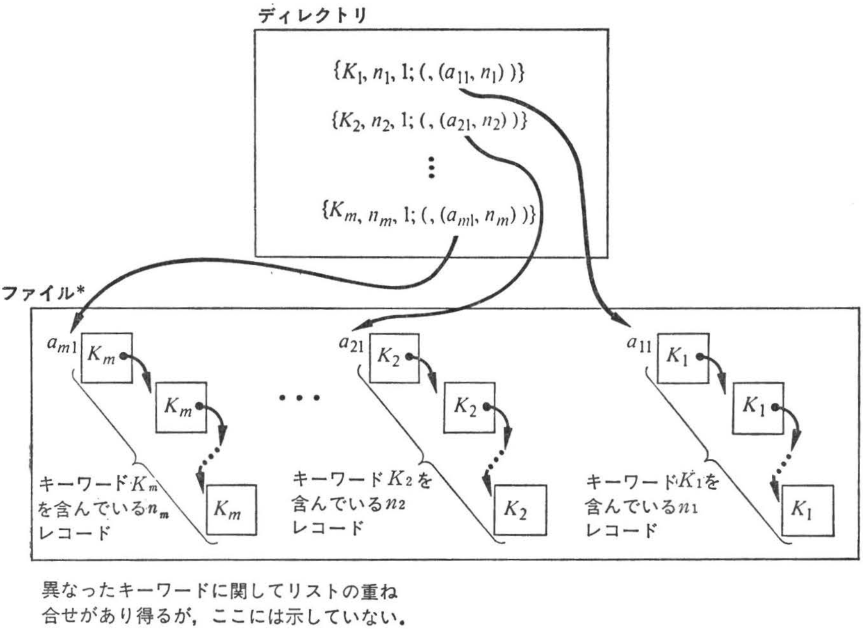


図 6.8　多重リストファイル

としたのは，ファイル内にはただ一つのセルしかなく，かつファイル内のすべてのレコードは同一のセル内にいることを意味する（図6.8）.

A. 多重リストの概念

多重リストファイルでは，ディレクトリ内の列がリストに対応する．各リストの先頭の番地だけが，ディレクトリにあらわれる．その後のレコードは，ポインタによって得られる．この例(図 6.9)のファイルに示したリストでは，一つ以上のレコードを共有することによって交差することを知るのは重要である．

"多重リスト" という術語は，レコード R が多重の K_i をもっているときは，必らず R はこれら K_i のリストの要素になっているところから生じてきた．上記の例では，レコード R' は2つのリストすなわち K_1 および K_2 リストの要素になっており，またレコード R'' は3つのリスト K_1, K_2 および K_3 の要素になっている．

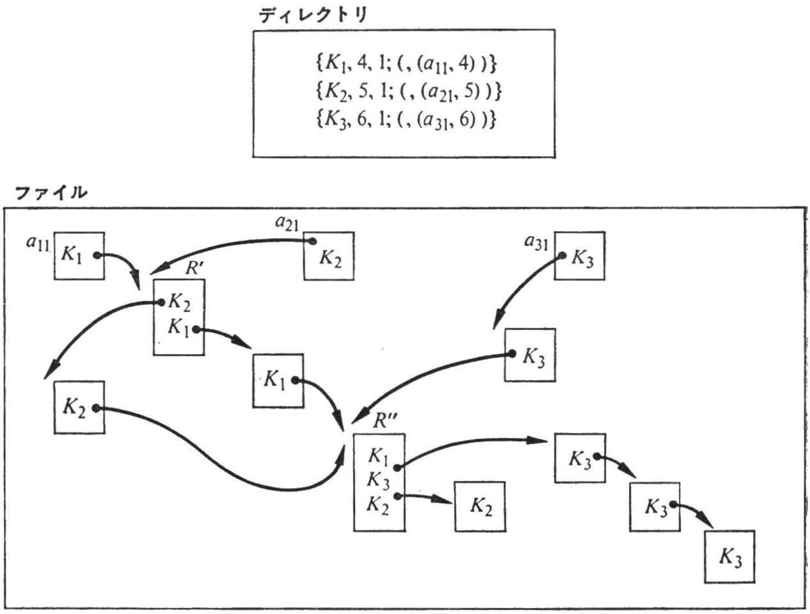


図 6.9　多重リストの実現

B.　多重リストファイルの利用

逆ファイルのディレクトリとは異なり，多重リストのディレクトリは，必ずしもファイル内のレコードに関する正確な統計的情報をもっているわけではない．たとえば，上記の例を見ると，K_1 に含まれているレコードは4個である．すなわち，$n_1 = 4$ である．また，$n_2 = 5$，$n_3 = 6$ となる．しかしながら，ファイル内のレコードの総数は，n_1, n_2 および n_3 の和よりも小さくなる．もっと詳細に言うと，ファイル内にはちょうど12個のレコードがあるけれども，n_i の和は15である．キーワードを含んだ質問に対して正確なレコードの数を得るためには，ファイルを探索しなければならない．これはもちろん，かなり費用のかかる操作である．

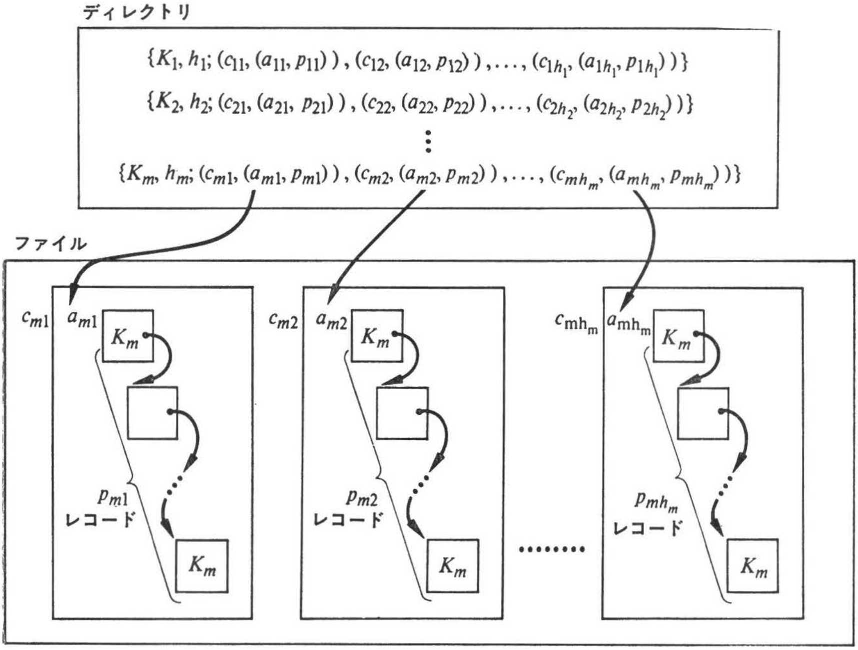
一方，レコード内のテキストを供給するのが主要な関数であるようなファイルがある．この場合，ファイル内のレコードの探索は義務づけられている．たとえば，技術論文の要約の文献的ファイルは，多重リストファイル構造として実現できる．ここでは，著者名，見出し語と句，および分類番号がキーワードとして使える．同一のキーワードのレコードは同一のリストに入っているので，著者，ある見出し語，ある分類で番号づけしたものなどに従ったすべての要約

のリストを得ることができる.

6.2.2.2.5 セル状多重リストファイル構造

セル状多重リストファイル(cellular multilist file)は, 各々のキーワード K_i に対し, セルの番号 $c_{i_1}, c_{i_2}, \cdots, c_{ih_i}$ が異なり, $0 < s < |F|$ が成り立つような一般化したファイル構造である (図6.10).

セル状多重リストファイル構造は, セルの概念を利用することを目指している. 一つのリスト内のレコードの数と一つのセル内のリストの数を制御することにより, セル状多重リストでは, セル内のほとんどすべての可能な記憶域をデータベース管理システムが利用することができる. 言い換えれば, システムはレコードのリストを, 記憶域の利用度を上げるためにセルにあてはめることができる. 一方, セルの番地とリストの先頭の番地とを保持してゆけば, セル状多重リストファイル構造では, データベース管理システムは記憶機器に対し



注: $i \neq j$ なる K_i-リストと K_j-リストの重ね合せと, $i \neq u$ に対して $c_{ij} = c_{uv}$ と等しくなることの可能性があるが, ここでは示していない.

図 6.10 セル状多重リストファイル

てもっと効率の良いアクセスをすることができる．後者については，アクセスのアルゴリズムに関する節で,かなり詳しく論ずるつもりである．したがって，ここでは，細かいことは述べないでおこう．

セル状多重リストファイル構造では，与えられたセル内では，どの2つのリストも，同じキーワードをもっていない．しかしながら，セルには，一つ以上のリスト(各々は，異なったキーワードをもつもの)がある．さらに，一つのレコードが複数個のキーワードで特徴づけられるならば，そのレコードは，キーワードに対するリストの中に入っている．

6.2.2.2.6 多様な応用に対する新しいファイル構造

セルの数，セル内のリストの数，およびリスト内のレコードの数を変えることにより，非常に多種類のファイル構造を作り出すことができる．特に，一つのキーワードに関しては索引乱構造で，もう一つのキーワードに関しては多重リストで，残りのキーワードに関しては逆ファイルという様に構造化したレコードをもつファイルを生成することができる．

応用に対してファイル構造を選択することを説明するために，ここに一つの例を示そう．ファイル構造を適正に選択すれば，応用に対して記憶域と処理とを大幅に節約できる．

出版物の要約の収集を使って，一般化したファイル構造として組織したファイルと，純粋の逆ファイル構造として組織したファイルとの比較をしてみる．記憶およびアクセスへの要求を推定する．この例は，いくつかの概念を詳しく解説することはせずに提示できるように，わざと簡単化している．

例 次のもので特徴づけできる30,000の出版物の要約（1要約当り約1000字)の収集.

- 分類番号(これらは，一般的な十進分類番号であると仮定する．一つの要約に対し，1個の主UDCと平均4個の二次UDCとがある)；
- 言語(50個の言語を仮定し，要約当り1個以上の言語がある．平均要約当り1.5言語.)；
- 出版社(300の出版社を仮定し，出版社に対し要約を平等に分布させる.)；
- アクセスの番号(出版物ごとに一つの一意的番号)；
- 要約(平均100語の可変長テキストから成る)；
- 著者(要約当り一人以上．一人の著者につき平均2つの要約があり,15,000人の名前がある).

例に対するユーザの処理の要求は，次の通りである．

1. 個々の要約は，一意的なアクセスの番号によって検索が可能でなければならない．
2. 与えられた言語で，かつ，特定の出版社から出版されたものの要約の番号の札がなければならない．
3. 2. の札で指定した要約の検索が高速にできること．
4. 一般十進分類番号による**すべての**要約のテキスト処理．

A. 提案する一般化したファイル構造

アクセスの番号に関しては，アクセス番号と要約との間に1対1の対応があるので，索引乱ファイル構造を選択する．

言語および出版者の属性に対しては，逆ファイル構造を選ぶ．これを選べば，同一の言語で書かれかつ同一の出版社から出版されたものは収集の中の小さい部分の本だけであろうから，ユーザは正しい札を得られ，本の要約の膨大な収集の中から速い検索の応答が得られる．

分類番号に関しては，要約が主 UDC 番号で整列され，二次 UDC 番号のリストがつながっている多重リストファイルの構造を推す．UDC 番号をたどることにより，一つの分類の中でテキストの処理のために1回に1個ずつ要約を検索することができる．

著者の属性に対しては，その処理の要求が指定されていない．しかし，著者当り平均2個の要約があるので，キーワード著者には，高々2個のリストをもつような構造を推薦する．

B. 任意に選んだファイル構造

前述したようにユーザの処理の要求を考慮せずに，この応用に対して逆ファイル構造を任意に選び，一般化したファイル構造と比較をしてみる．

ACCESSION NO.（アクセス番号），AUTHOR（著者），LANGUAGE（言語）および PUBLISHER（出版社）の属性とその値とに対するディレクトリの記憶域の要求は，同じである．しかし，PRIME CLASS. NO.（主分類番号）および SEC. CLASS. NO.（二次分類番号）の属性に対しては，追加の記憶が必要となる．

C.　ディレクトリの記憶域とアクセスの要求とから見た2つのファイル構造の比較

A. で提案した一般化したファイル構造と，**B.** で任意に選んだ純粋の逆ファイル構造とに対する総合的な計算について，以下に示す.

1.　キーワードの属性に対する仮定は，次の通りである.

主分類番号: 要約ごとに一つの番号があり，値ごとに5個の要約がある．したがって，6000個の異なった値がある.

二次分類番号: 要約ごとに平均4個の番号があり，値ごとに20個の要約がある．そこで，6000個の異なった値がある.

言語: 50個の異なった言語があり，要約ごとに平均1.5個の言語がある.

出版社: 要約は出版社に対して平等に分布している．それゆえ，出版社ごとに，100個の要約がある.

アクセスの番号: 要約ごとに一つの番号があり，値ごとに一つの要約がある.

著者: 15,000個の異なった名前があり，著者ごとに約2個の要約がある.

2.　一般化したファイル構造に対するディレクトリの記憶域.

属性 ACCESSION NO. に対して，索引乱ファイル構造は，次のものが必要となる．この属性名に対し，10文字(すべての属性に対して10文字で打切る)，
その制御情報に対して2文字，
値に対して10文字，
その制御情報に対して4文字，
リストの先頭(のレコード)の番地に対して6文字.
したがって，

$$(10+2)+(10+4+6)\times 30,000 = 600,012\,\text{文字} = \frac{600,012}{2K}$$

$\cong$ 300物理レコード（物理レコードの大きさは $2K$ 文字と仮定する).

同様にして，属性 LANGUAGE に対して，逆ファイル構造は，次のものが必要となる.

$(10+2)+50\times(10+4)+1.5\times6\times30{,}000=270{,}712$ 文字
$\cong 136$ 物理レコード.

属性 PUBLISHER と 300 個の出版社に対して，次のものが必要である.
$(10+2)+300\times(10+4)+6\times30{,}000=184{,}212$ 文字
$\cong 93$ 物理レコード.

属性 AUTHOR と 15,000 の異なった名前に対して，次のものが必要となる.

$(10+2)+15{,}000\times(10+4)+6\times30{,}000=390{,}012$ 文字
$\cong 195$ 物理レコード.

属性 PRIME CLASS. NO. に対して，多重リストファイル構造では，次のものが必要である.

$(10+2)+(10+4)+6=32$ 文字
<1 物理レコード.

属性 SEC. CLASS. NO. に対して，多重リストファイル構造では，次のものが必要である.

$(10+2)+(10+4)+6\times6{,}000=36{,}026$ 文字
$\cong 36$ 物理レコード.

したがって，ディレクトリの記憶に必要な総和は，次の通りである.
$300+136+93+195+1+36=761$ 物理レコード.

3. 純粋な逆ファイル構造に対して必要な追加の記憶域.
PRIME CLASS. NO. に対して，次のものが必要である.

$(10+2)+6{,}000\times(10+4)+6\times30{,}000=264{,}012$ 文字
$\cong 132$ 物理レコード.

SEC. CLASS. NO. に対して，次のものが必要である.

$(10+2)+6{,}000\times(10+4)+6\times4\times30{,}000=804{,}012$ 文字
$\cong 402$ 物理レコード.

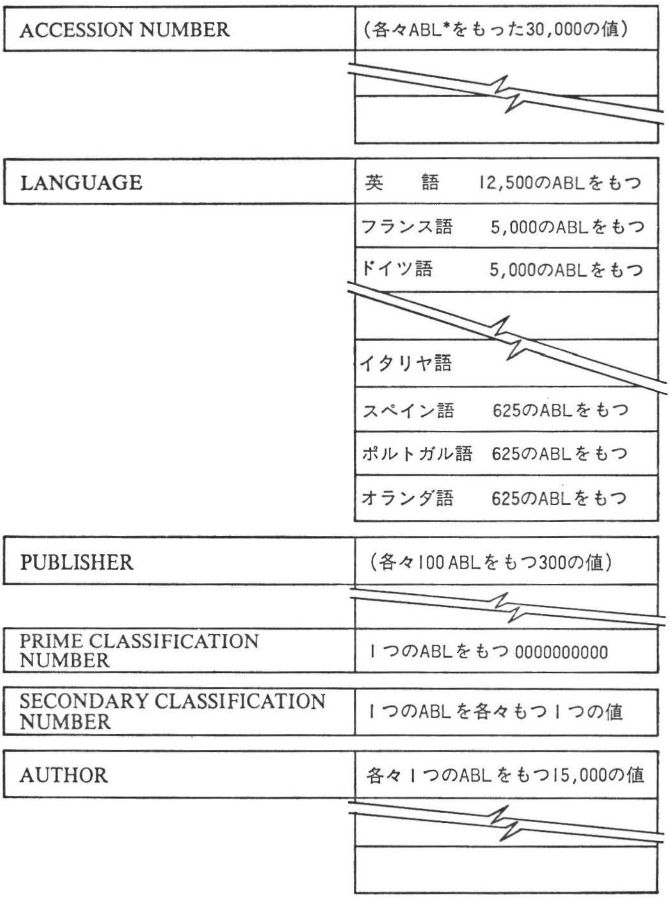
したがって，追加の記憶域の必要量は，$132+402=534$ 物理レコードである．すべて逆転したファイルに対するディレクトリの記憶域の必要量の総和は，

$$817+534=1,351 \quad \text{物理レコード}$$

である．

4.　アクセスに対する要求の総計算量

これらの計算では，ディレクトリを構成する物理レコードにアクセスする回数に関心がある．当然これらの数は，ディレクトリの特定の実現性に敏感に反応する．しかしながら，ここは，ディレクトリのアクセスの



*ABLとは，リストの先頭(になるレコード)の番地をあらわす。

図 6.11　一般化したファイル構造のディレクトリの配置

見地から，ディレクトリのいろいろな実現を比較するところではない．説明したいことは，与えられた実現の中で，多様な問い合せの型に関してのアクセスの数である．これらの数は，実現によって変化するかもしれないけれども，お互いにほとんど同じものとなると信じている．

LANGUAGE = ENGLISH および PUBLISHER = A. B. DOE

(ユーザの処理の要求の2．および3.) という問い合せの型に対して，英語の要約を含んでいるものはおよそ38物理レコード(6 × 12,500 = 75,000文字 = 38物理レコード)あり，また，指定した出版社を含むものは1レコード(300出版社は93レコードに分布している)ある．したがって，レコードのアクセスの最大数は，38 + 1 = 39である．この推定は，一般化したものおよび純粋の逆ファイルの両者に対して成り立つ．

主分類番号を含むすべての要約のテキストの処理 (ユーザの処理の要求の4.)に対して，これらの要求をファイル内で探索するのは両方の構造で同じであることに注意しよう．2つの構造の異なる点は，要約の番地を決定するためのディレクトリに含まれる物理レコードの数である．純粋な逆ファイル構造の場合には，要約の番地を含むディレクトリには，132物理レコードがある．それゆえ，これらのレコードを使って，要約の番地を決めなければならない．一方，一般化したファイル構造では，要約の番地と“最小”の二次の一般十進分類番号とを含むレコードは，37物理レコードしかない．したがって，この場合のディレクトリのアクセスは最小になっている．

ACCESSION NO. = *nnnnnnnnnnnn* という問い合せの型に対して，一般化したファイルおよび純粋の逆ファイルのどちらのディレクトリでも，一つのレコードだけしかないことに注意しよう．ディレクトリを使えば，ディレクトリの中を順番に探索しなくても直接このレコードを検索できるので，各ファイル構造に対して，アクセスの要求は一つの物理レコードである．

要約すると，以下のような推定(図6.12)を得る．

<table>
<tr><th colspan="2"></th><th>一般化したファイル構造
（1個の索引乱構造,
2個の多重リスト,
3個の逆ファイル）</th><th>逆ファイル
構造だけ</th></tr>
<tr><td colspan="2">ディレクトリの記憶域への要求
（物理レコードの数）</td><td>761</td><td>1,351</td></tr>
<tr><td rowspan="3">ディレクトリのデコードに必要な物理レコードの最大数</td><td>LANGUAGE＝ENGLISH
および
PUBLISHER＝A. B. COX
の問い合わせ型に対して
（ユーザの処理要求2および3）</td><td>39</td><td>39</td></tr>
<tr><td>主分類番号を含むすべての
要求・テキスト処理に対して
（ユーザの処理要求4）</td><td>37</td><td>132</td></tr>
<tr><td>ACCESSION NO.＝
0123456789という問い
合わせ型に対して
（ユーザの処理要求1）</td><td>1</td><td>1</td></tr>
</table>

図 6.12

6.3 問い合わせ

キーワード K は，レコード R に対して $K \in R$ のとき**真**(true)であるという．（ファイル F に関する）**問い合せ**(query)とは，（F 内のレコードを特徴づける）キーワードの論理（および算術）式によって与えられる命題である．問い合せは，レコード R に対してこの命題が成り立つとき真であるという．この場合，R は問い合せを**満足する**という．我々の議論では，すべての問い合せは，次のような和標準形(DNF)になっていると仮定する．

$$(K_{11} \wedge K_{12} \wedge \cdots \wedge K_{1r_1}) \vee (K_{21} \wedge K_{22} \wedge \cdots \wedge K_{2r_2}) \vee \cdots \vee (K_{t_1} \wedge K_{t_2} \wedge \cdots \wedge K_{tr_t}).$$

ここに，$\wedge$ は **AND**，$\vee$ は **OR** をあらわす．
たとえば，次の問い合せは，きちんと定義された DNF である．

（行番号，1102. 1)，
（従業員，John Doe) $\wedge$ （給与，15,000)，

((年, 1964) ∧ (月, 4月)) ∨ ((年, 1964) ∧ (月, 5月) ∧ (時刻, 11: 00 AM)).

第一の問い合せは，一つのキーワードだけをもった和標準形である．第二の問い合せは，一つの積だけをもつ和標準形である．この積では，2つのキーワードがある．最後の問い合せは，2つの積から成り，一方は2つのキーワード，もう一方は3つのキーワードをもつ．

次に示すようなレコードに対するユーザの要求を考えよう．

(主題, 計算機) ∧ ((領域, 構成) ∨ (細分領域, アーキテクチャ)).

最初，レコードに対するユーザの要求を，和標準形の形式となるキーワードの論理(および算術)式に変換する．DNF を入力として，ディレクトリのデコーディング手続きがディレクトリのエントリを処理し，ファイルの探索関数によって以後使うための必要な情報を作り出す．

キーワードの論理(および算術)式は，ディレクトリのデコーディングとファイルの探索に対する問い合せとして使えるだけでなく，他の多くの目的のためのレコードの集合の仕様としても使える．このために，時々，DNF を簡略化して**式**または**記述**と呼ぶ．

6.4 ディレクトリのデコーディングとファイルの探索

ファイルのディレクトリ内で，任意の列

$$\{K, h;\ D(K)\}$$

を考えよう．ここで $D(K)=\{(c_1, (a_1, p_1)), (c_2, (a_2, p_2)), \cdots, (c_h, (a_h, p_h))\}$．

次に，ディレクトリから情報を引き出すいくつかの操作子を導入しよう．

セルエントリ操作子(cell entry operator) d は，与えられたキーワード K に対して，記憶セル c の中のレコードの数を与える．与えられたキーワード K に対して，記憶セル c に付随して $D(K)$ 内に1個以上のエントリ $(c_j, (a_j, p_j))$ があってもよいことに注意しよう．それゆえ，この操作子は，c_j が c と同じであるような p_j をすべて数え上げる．記号的には，次のように書く．

$$d(K, c)=\sum_j p_j \qquad ここに (c_j, (a_j, p_j)) \in D(K),\ c_j=c.$$

以後，キーワード K と c におけるセルに対応して，この和を P_j と呼ぼう．それゆえ，与えられたセルとキーワードとに対して d を適用すると，同一の

セルとキーワードとに対していろいろのエントリに含まれているレコード数の和を生成する．

セル識別名操作子(cell identifier operator) b は，与えられたキーワードに対して付随している記憶セルを決める．記号的には，次のように書く．

$$b(K) = \{c|(c, (a, p)) \in D(K)\}.$$

ディレクトリ探索操作子(directory search operator) f は，与えられたキーワードと記憶セルとに付随した先頭番地 a_{ij} を決める．記号的には，次のように書く．

$$f(K, c) = \{a|(c, (a, p)) \in D(K)\}$$

f, b, d を適用してキーワードのすべての先頭番地 a_{ij}，それに付随したセルの番地，および各セル内のレコードの数を決めれば，それを**デコード**したという．与えられたキーワードをデコードするために上記の操作を適用するには，セル識別名操作子を最初に適用して，そのキーワードに付随しているすべてのセルの番地の集合を求めなければならない．次に，この集合の各々の番地に対して残りの2つの操作子を適用して，そのキーワードに対してセルの中のレコードの数と，そのキーワードのリストに対する先頭番地とを求めればよい．これらの操作子の使用については，アクセスアルゴリズムを議論するときに，また述べよう．

上述したディレクトリデコーディングの操作子のほかに，与えられたファイル F に対して**ファイル探索関数** (file search function) とよぶ関数 g を使うと便利である．g の領域は，F 内のすべてのキーワードの集合 K と F 内のすべての番地の集合 A の直積 $K \times A$ であるのに対し，g の値域は A である．実際に，g は，各々のリストにおいて先頭番地からそれに続くポインタをどのようにたどるのかを教えてくれる．記号的には，次のように書く．

$$y = g(K, x).$$

ここに，y は番地が x であるようなレコードの K ポインタである．完全性のために，y が空の K ポインタのとき，$y = 0$ と設定する．直観と合致させるために，レコード R は，R の K ポインタが g によって生成されたものであるとき，**検索された**あるいは**アクセスされた**という．レコードは，その番地がポインタを作成するために g によって使われたときにだけ，検索される．

たとえば，図6.1に示したレコードをもう一度使ってみよう．最初に，g を R と次の一つのキーワードに適用する．R の番地は001なので，$g(K_6, 001) =$

101，すなわち R' の番地を得る．次に，g を R' と同じキーワード K_6 に対して適用し，$g(K_6,\ 101) = 213$，すなわち，R'' の番地を得る．最後に，$g(K_6,\ 213) = 0$ となるが，これは，K_6 リストにはこれ以上レコードがないことを示している．この場合，与えられた R から，R' と R'' とが検索された．

次に示すように，g を R のほかのキーワードに適用することによって，残りのレコードの検索も可能である．

$$
\begin{aligned}
&g(K_1,\ 001) = 0\\
&g(K_2,\ g(K_2,\ 001)) = g(K_2,\ 003) = 0\\
&g(K_3,\ g(K_3,\ 001)) = g(K_3,\ 008) = 0\\
&g(K_4,\ g(K_4,\ g(K_4,\ 001))) = g(K_4,\ g(K_4,\ 004))\\
&\qquad = g(K_4,\ 204) = 0\\
&g(K_5,\ g(K_5,\ g(K_5,\ 001))) = g(K_5,\ g(K_5,\ 011))\\
&\qquad = g(K_5,\ 204) = 0\\
&g(K_6,\ g(K_6,\ g(K_6,\ 001))) = g(K_6,\ g(K_6,\ 101))\\
&\qquad = g(K_6,\ 213) = 0
\end{aligned}
$$

上の例で，関数 g によって検索されるレコードは，番地 001，003，004，008，011，101，204 および 213 である．番地 204 におけるレコード R''' は，2つのキーワードを含んでいるので，それは2度検索されてしまった．

効率の良いデータベース管理の関数は，システムが与えるディレクトリのデコーディング手続きとファイルのアクセスの戦略の能率にかなりの程度まで左右される．これらのアルゴリズムにより，ユーザが要求したときに，望みのレコードに高速にアクセスできないならば，システムのデータベース管理の関数を効率良くすることはほとんど不可能である．このため，データベース管理の関数を支援するためには，システムは知能の高いディレクトリデコーディングの手続きと効率の良いファイルアクセスの戦略を採用しなければならない．

より効率の高い手続きおよび戦略と，効率の低いものとを比較するために，一般化したファイル構造の中ですべてのレコードにアクセスするための2つのアルゴリズムを形式的に論じよう．

6.5 2つのアクセスアルゴリズム

ここで，単一のキーワード K_i を含むようなファイル F 内のすべてのレコードにアクセスするアルゴリズムを述べよう．次に，キーワードを使った式を満たすような F 内のすべてのレコードにアクセスするアルゴリズムを述べよ

う．最後に，これらの2つのアルゴリズムの違いを研究しよう．

6.5.1 順次アクセスアルゴリズム

F 内の K を含むすべてのレコードは，次の手続きによってアクセスできる．

1. K がデコードされるまで，セル識別名，セルエントリおよびディレクトリ探索の操作子(b, d および f)を適用する．この結果，リストの先頭番地 h 個の列 Ψ が得られる．
2. Ψ 内の一つの先頭番地 a に，先ずファイル探索関数 g を適用する．セルは，K リスト内に p 個のレコードを含むものと仮定する．次に，K を含む p 個のレコードが検索されるまで，2. でみつかるポインタの各々にファイル探索関数 g を繰り返し適用する．
3. Ψ から a を取り除く．Ψ が空になるまで，2. を続ける．

このアルゴリズムは，**リストの順次処理**とよばれる．直観的な説明は，次の通りである．与えられたキーワードに対して，最初にそのリストの先頭番地をすべて見つけ出す．そのようなリストを一つ選び，リスト内のレコードがなくなるまで，先頭のレコードからそれに続くレコードをずっとたどってゆく．このあと，もう一つのリストを選び，上に述べたような方法でこれをたどる．これを，リストがなくなるまで続ける．一つのリストをたどっている間，別のリストをたどることは始められない．それゆえ，リストは順番に，すなわち順次的に処理されなければならない．

図6.13に示すような単純なファイルを考えよう．このファイルのすべてのレコードを検索することにより，順次アクセスアルゴリズムの使い方を示そう．ディレクトリには，たった3個のキーワードしかないので，次に与えるようなユーザの問い合せに基づいてレコードを検索することを示すことになる．

$$K_1 \vee K_2 \vee K_3.$$

A.　ここで，K_1 に対して順次アクセスアルゴリズムを適用しよう．

ステップ 1.　セル識別名操作子 b を適用し，このキーワードに対し記憶セルの番地00を得る．セルエントリ操作子 d を適用し，キーワード K_1 に対しそのセル内のレコードの数を知る．もっと詳しくいうと，

$$n_1 = p_{11} = 5.$$

ディレクトリ探索操作子 f を適用し，K_1 に対する先頭番地をすべて決める．すなわち，

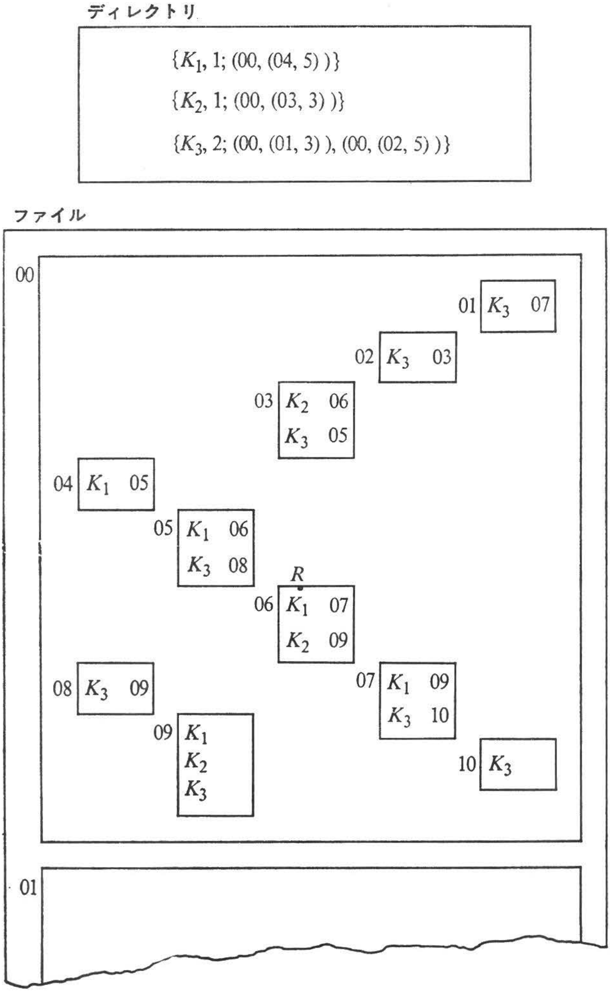


図 6.13 単純なファイルの例

$$a_{11} = 04.$$

この結果，列 Ψ はただ一つの先頭番地からできる．

$$\Psi = (04).$$

ステップ 2. ファイル探索関数 g を Ψ に適用する．

$$g(K_1, 04) = 05,$$
$$g(K_1, 05) = 06,$$
$$g(K_1, 06) = 07,$$
$$g(K_1, 07) = 09,$$
$$g(K_1, 09) = 0.$$

ステップ 3. 04をΨから取り除く．ここで，Ψは空となる．したがって，K_1を含むすべてのレコードは検索されてしまった．実際，ステップ2.で番地が見つけられたようなレコードは5個である．

B. 同様にして，キーワードK_2に対しても，順次探索アルゴリズム(すなわち，3つのステップ)を繰り返す．この結果，K_2を含むレコードが3個検索される．

C. 最後に，キーワードK_3に，順次探索アルゴリズムを適用する．K_3リストが2つあるので，列Ψは2個の先頭番地を含む．

$$\Psi = (01, 02).$$

Ψ内の最初の番地にファイル探索関数gを適用すると，次のものを得る．

$$g(K_3, 01) = 07,$$
$$g(K_3, 07) = 10,$$
$$g(K_3, 10) = 0.$$

ここで，Ψ内の2番目の番地を使う．

$$g(K_3, 02) = 03,$$
$$g(K_3, 03) = 05,$$
$$g(K_3, 05) = 08,$$
$$g(K_3, 08) = 09,$$
$$g(K_3, 09) = 0.$$

すなわち，8個のレコードが検索された．

要約すると，この例は，このアルゴリズムの単純さを示しているだけでなく，順次アクセスの限界をも指摘していることに気が付く．

ファイルから問い合せを満足する10個のレコードを検索するためには，このアルゴリズムでは16回($5+3+8=16$)のレコードアクセスを行なった．問い合せを満足するレコードの数と，ファイル探索の間アクセスされるレコードの数との比を**精度**(precision)と定義すると，

$$精度 = \frac{10}{16} = \frac{5}{8}$$

となり，これは，順次探索アルゴリズムは精度が良くないことを示している．明きらかに，精度の値が1のときにのみ，絶対的な精度が実現できる．

一番低い番地をもつセルに属している先頭番地 a に対して常にファイル探索関数 g を適用していれば，このアルゴリズムがいくらか改良できる．すなわち，セル内のすべの K のリストが，他のセル内の K リストが探索される前に，順次的に処理される．さらに，セルは，低いセルの番地から高いものへと順次的にアクセスされる．しかしながら，大きく改良するためには，次に示すアルゴリズムを導入する．

6.5.2　並列アクセスアルゴリズム

ここで，一般化したファイル F 内で，キーワードの和標準形であらわされるような与えられた問い合せを満足するレコードをすべてアクセスするアルゴリズムを形式化しよう．問い合せを，次のように書く．

$$(K_{11} \wedge K_{12} \wedge \cdots \wedge K_{1r_1}) \vee (K_{21} \wedge K_{22} \wedge \cdots \wedge K_{2r_2}) \wedge \ldots$$
$$\vee (K_{t1} \wedge K_{t2} \wedge \cdots \wedge K_{tr_t}).$$

ここで，与えられた問い合せには t 個の項があり，i 番目の項には r_i 個のキーワードがある．

1.　セルの番地を決定する．

t 個の項各々に対して，セル識別名操作子 b を繰り返し適用して，項の r_i 個のキーワードをすべて処理する．この結果，セルの番地の集合 $\theta_{ij}(j=1, \cdots, r_i)$が得られる．これらの集合の共通部分を作る．

$$\theta_i = \bigcap_{j=1}^{r_i} \theta_{ij}.$$

集合 θ_i に番地が含まれているセルに入っていないようなレコードは，問い合せの i 番目の項を満足していない(これらのレコードは，i 番目の項の一つのキーワードは含んでいないからである)．したがって，$i=1, \cdots, t$ に対して θ_i のセルに入っているレコードだけが，さらに処理をすすめる対象となる．

2.　主キーワードを選択する．

問い合せの i 番目の項

$$(K_{i1} \wedge K_{i2} \wedge \cdots \wedge K_{ir_i})$$

とそのセルの番地の集合 θ_i を考える．

議論を簡単化するために，添字 i を除いて，項とそのセルの番地の集合とをそれぞれ次のように書く．

$$(K_1 \wedge K_2 \wedge \cdots \wedge K_r) \quad \text{および} \ \theta.$$

このように簡略化すれば，ステップ1．の式は，

$$\theta = \bigcap_{j=1}^{r} \theta_j$$

となる．これは，すべての $c_v \in \theta$ に対し，次のことを意味する．

すべての u に対し

$$(c_v, (a_{uv}, P_{uv})) \in D(K_u).$$

言い換えれば，c_v は，K_u だけでなく，項の他のすべてのキーワードによっの特徴づけられるレコードを含むセルの番地である．

θ 内の各々のセル番地 c_v と項の各々のキーワード K_u に対して，セルエントリ操作子 d を繰り返し適用する．この結果，異なったキーワード K_u とセル番地の対に対する和$(d(K_u, c_v) =)P_{uv}$ の集合が得られる．和 $d(K, c)$は，c のセルにいるキーワード K を含んでいるレコードの数であることを思い出そう．

各セル c_v に対し，最小の P_{uv} を選ぶ．対応するキーワードは，そのセルに対する**主キーワード**(prime keyword)という．一つのセルの主キーワードは，別のセルのキーワードとは異なってもよいことに注意しよう．それにも拘わらず，これらの主キーワードは，問い合せの i 番目の項のキーワードであり，したがって，i 番目の項を満足する任意のレコードを特徴づける．

セルの主キーワードは，同じ積形式(すなわち，問い合せの i 番目の項)の中で他のキーワードのリストに比べの最小のリストをもつキーワードであるから，このキーワードは，セル内で以後行なわれるファイル探索に対して，積形式のすべてのキーワードの代表として使われる．積形式を満足する大きいリストの代りにその積形式を満たす可能性のある最小のレコードのリストをアクセスするので，この方式は，明きらかに精度を向上させる．しかし，この積形式を満足するレコードを含んでいるセルは，多数ある．したがって，各セルに対して主キーワードを持たなければならない．

この情報から，次の形式の対の集合 γ' を形づくる．

(K_v', c_v)　　ここに，K_v' は番地 c_v のセルに対する主キーワードであり，

θ 内のすべてのセルの番地に一つづつそのような対がある．

項に対して γ' を得ると，問い合せ内の各項に対して処理を繰り返し，その結果，問い合せの各項 t に対し γ_i' が得られる．これらの γ_i', γ の和集合をディレクトリのデコードに使う．

3．ディレクトリのデコード．

γ 内のすべての対(K', c)に対して，ディレクトリ探索操作子 f を適用して，検索の間に探索すべきレコードのリストの先頭番地を得る．集合 Ψ に f を完全に適用したものに対応する番地の列として，Ψ を初期設定する．Ψ は，最初，K リストの先頭番地が入っていることに注意しよう．アルゴリズムのステップ6．では，K ポインタが Ψ に加えられ，ステップ4．では，Ψ から番地が取り除かれる．検索過程で使用される最初空の集合を E としよう．

4．ファイルの探索．

Ψ 内で番地が最初にあらわれたら，それ以後そのすべての出現を取り除いて得られる Ψ の単調部分列を σ とする．

σ 内のすべての番地がファイル探索関数 g によって使われてしまえば，探索は完了し，アルゴリズムは，ステップ7．へ続く．そうでなければ，σ 内で g によってまだ使われていない最初の番地を選び，それを σ と呼ぶ．また，この番地を Ψ から消してしまう（もしも，σ が単調増大であれば，この番地は，σ 内で最小の番地である．もしも，σ が単調減少であれば，この番地は最大のものである）．

a は，K リストの先頭番地か，K ポインタかのどちらかであることに注意しよう．どちらの場合にも，a はレコード R の番地である．レコード R を含むセルに対する主キーワードであるすべてのキーワード $K \in R$ に対して，g を a と K に適用して，K ポインタの列 σ_1 を得る．すなわち，$K \in R$ が g によって使用される前に，対(K, c_a)は γ の中になければならないが，ここで，c_a はレコード R を含むセルの番地である．

a におけるレコードが検索されたことに注意しよう．σ_1 から，すべての空ポインタを除く．

5．レコードを調べる．

このアルゴリズムの目的は，与えられた問い合せを満足するような F 内のすべてのレコードの検索であることに注意して，ちょうど検索してきたレコードを調べて，そのレコードが問い合せを満足するかどうかを決める．そうであれば，それを集合 E の中に含める．そうでなければ，それを E には含めない．

6.　次のレコードを検索するための準備をする.

σ_1 のすべての番地を Ψ に移す．ここで，アルゴリズムをステップ **4.** から繰り返す.

7.　検索の完了.

E 内には，与えられた問い合せを満足するすべてのレコードがある．

このアルゴリズムは，**リストの並列処理**(parallel processing of lists) と呼ぶ．これは，一つのリストをたどっている間に，別のリストをたどることもできるという点で，順次処理と違っている．言い換えれば，複数個のリストは，どれも完了することなしにたどってもよい．本質的には，これらのリストは並列に処理されている．もっと詳しく言うと，このアルゴリズムは，問い合せ内で，項ごとに2個以上の主キーワードを使って探索できる．実際，項ごとに，**記憶セルにつき**一つの主キーワードを使って並列的に探索している．

重要な点

このアルゴリズムの重要な点について，いくつか強調しておこう．

A.　このアルゴリズムのステップ **1.** では，一つの積形式で個々のキーワードいくつかを満たすけれどもキーワード全体の積形式は満たすことのないようなレコードをもつ不必要な記憶セルをスキップするように設計されている．

B.　一つの積形式にあらわれるキーワードの集合の中では，検索のために選択されるレコードの数は，出来る限り小さくしている．

与えられた記述の項(積形式)を考えよう．

$$(K_1 \wedge K_2 \wedge \cdots \wedge K_r).$$

先頭番地が a_{ij} でセル番地が c_{ij} であるような K_i リストは L_{ij} とし，同一のセル内の K_i リストの和集合を L_i としよう．このとき，次式が成り立つ．

$$L_i = \bigcup_{j=1}^{w} L_{ij}, \qquad i = 1, 2, \cdots, r; \quad w \leqq h_i.$$

各々の L_i は，ちょうど $\sum_j p_{ij}$ レコードを含んでいる．上記の積形式を満たすレコードの集合 S は，実際には，L_i の共通部分である．すなわち，

$$S = \bigcap_{i=1}^{r} L_i = \bigcap_{i=1}^{r} \bigcup_{j=1}^{w} L_{ij}.$$

さらに，すべての i に対し，S は L_i の部分集合である．しかし，これらの

L_i は，大きさが変動してもよい．小さな L_i に比べて大きな L_i から S を得るためには，より多くの労力が必要となるのは，あきらかである．S を得るための労力を最小にするためには，最小の L_i の K リストを介してたどるべきである．主キーワードを選択すると，レコードのアクセスのための問い合せの各々の積形式に対して最小のレコードの集合を決めることができる．

これらのリストを決めるのに際し，これらのリストが入っているセルも考慮していることを，もう一度述べておく．判断の基準は，セル毎という基本に従って，最小のリストを選ぶことである．それゆえ，与えられた積形式に対して，与えられたセルにおいてただ一つのたどるべきキーワードに対するリストがあり，それらのリストは最小のレコード数のものである．これらのキーワードリストのキーワードは，主キーワードであるが，これはセル毎に異なっている．それにも拘わらず，これは積形式のキーワードである．ステップ2. は，検索のために，満足すべきレコードの最短のリストを決めるように設計してある．

C.　図6.13を見てわかるように，(06における)レコード R は，キーワードを2個持っているので，2回検索される．一般に，レコードは多数のキーワードを持っていてよい．したがって，すべてのキーワードのリストをたどると，たった一つだけの検索が必要なのにもかかわらず，そのレコードは何回も検索されるであろう．これは，特に，順次アクセスアルゴリズムを使うとあきらかになる．並列アクセスアルゴリズムでは，ファイルの探索において，Ψ のかわりに番地列 σ を使い，ファイル探索関数のためには使われていなかった番地 a を σ の中から選ぶので，一回アクセスされたレコードはもう一度アクセスされることはない．この結果，検索の労力は大幅に節約される．

D.　ステップ4. で番地を整列し，冗長な番地を除いていくと，記憶機器の可動アクセス機構は機器の一つの端からもう一方の端へ走査することができ，再位置づけが最小で済む．ステップ4. の番地の列が単調であることに依存して，アクセス機構はどちらか一方に走査する．その列が単調減少であれば，その機構は“高番地の端”から“低番地の端”へ走査をする．その列が単調増加であれば，低い方から高い方へ走査する．一方，順次アクセスアルゴリズムではリストに対するアクセスを始めるごとに，再位置づけを引き起こすかもしれない．

E.　ステップ5. では，検索してきたレコードの集合 F は，問い合せを満足するレコードの集合 E を真に含んでいてよい(すなわち，F の中には，E に属し

ていないレコードがあってもよい)．しかしながら，逆ファイル構造の場合，ステップ1．で行なった共通部分の作成により，2つの集合 F と E とは同一になる．したがって，この場合検索精度は絶対的である．

F. このアルゴリズムは，ただ一つの問い合せ，すなわち，一時に一つのキーワードの和標準形だけを扱っているけれども，これは一括処理式の問い合せにも簡単に拡張できる．この拡張としては，問い合せの待ち行列を加えること，複数個の問い合せを(結合した)和標準形に直すこと，および，ユーザの問い合せに対応した結合した和標準形の一部に基づいて条件を満足しているレコードをユーザに配ることがある．待ち行列は，問い合せを保持しておくために使う．待ち行列に入っている各エントリは，和標準形に変換される．待ちの期間の最後には，次の形式のような結合した和標準形(DNF)を作る．

$$(\mathrm{DNF})_1 \vee (\mathrm{DNF})_2 \vee \cdots \vee (\mathrm{DNF})_n.$$

この場合，n 人のユーザがおり，$(\mathrm{DNF})_i$ は，ユーザ i の和標準形をあらわしている．あきらかに，和標準形の和は，やはり和標準形である．並列アクセスアルゴリズムが入力として使うのは，この結合したDNFである．条件を満すレコードが検索されると，そのレコードはDNFを満さなければならない．もっと詳しく言えば，このアルゴリズムは，ある i に対してこのレコードが $(\mathrm{DNF})_i$ を満していることを示している．このレコードをユーザ i に分配することは，直ぐにできるであろう．このような拡張は，いろいろのユーザが中央のデータベースに対して多数の問い合せをするようなオンラインあるいは時分割システムの環境では，特に役に立つ．

G. 並列アルゴリズムは，ネットワークや階層的ファイル構造に対する問い合せを扱うように拡張できる．最初に，一般化したファイル構造をこれらの構造に拡張することから論じよう．基本的には，階層の各レベルにはディレクトリがある．階層の“根”には，(主)ディレトクリがある．ディレクトリのエントリは，キーワード K_i^1，リストの数 h_i^1，および，セルの番地 c_j^1，リストの先頭番地 a_{ij}^1，リスト内のレコードの数 p_{ij}^1 の3つ組から構成される．a_{ij}^1 は，勿論，リストの第1レコードの番地である．このレコードは，ディレクトリであっても，そうでなくてもよい．そのレコードがディレクトリであれば，ファイル構造のレベルが追加されたことになる．すなわち，番地 a_{ij}^1 において，レベル2のファイルのディレクトリを見出すことができる．同じ規則により，図6.14に示すように，ファイル構造の $(n-1)$ レベルでディレクトリがあれば，n レ

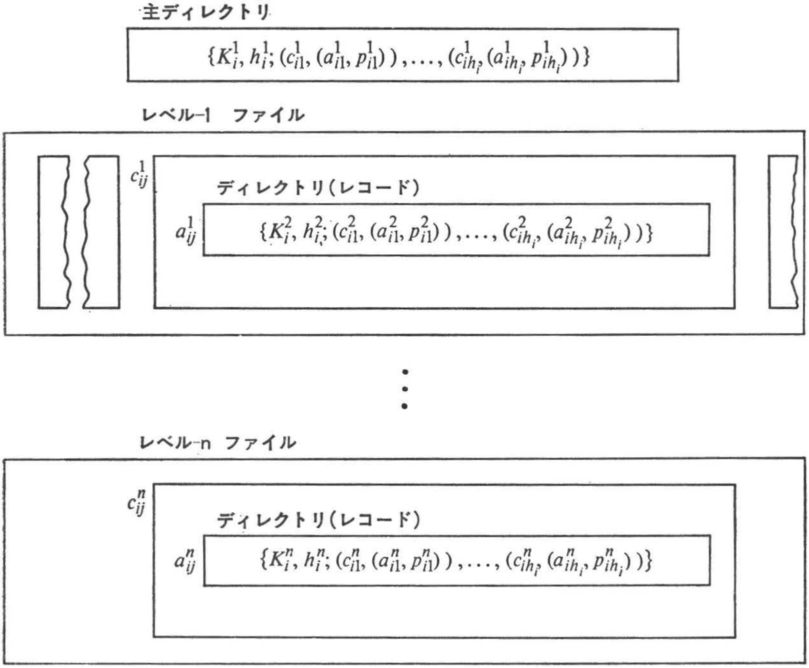


図 6.14　階層的ファイル

ベルのファイル構造が作られる．

レベルnに於けるレコードの集合を見つけるために，システムはレベル1でディレクトリのデコードとファイルの探索とを行なう．ファイルの探索は，レコードが次のレベルのファイルのディレクトリであるとわかったとき，成功したという．再び，新しいレベルのディレクトリを使って，ディレクトリのデコードとファイル探索の手続きを続ける．遂には，$(n-1)$レベルに対するディレクトリが見つかり，それを使うとnレベルのレコードの集合を見つけ出すことができる．

ネットワークおよび階層のファイル構造に対しては，問い合せは，キーワードの多重レベルの論理式となる．各レベルにおいて，そのレベルに対するキーワードの論理式を使う．そのレベルにおいてその論理式を満すレコードが一つもない場合は，次のレベルでディレクトリのデコードとファイル探索とを進める必要はない．言い換えれば，式$(K_{11}^m \wedge K_{12}^m \wedge \cdots \wedge K_{1r_1}^m) \vee \cdots \vee (K_{t_1}^m \wedge K_{t_2}^m$

$\wedge \cdots \wedge K^{m}_{tr_t})$を満たすレコードがなければ，式$(K^{m+1}_{11} \wedge K^{m+1}_{12} \wedge \cdots \wedge K^{m+1}_{1s_1}) \vee \cdots \vee (K^{m+1}_{v1} \wedge K^{m+1}_{v2} \wedge \cdots \wedge K^{m+1}_{vs_u})$に対してディレクトリのデコードやファイル探索を行なう必要はない．

上記の探索戦略は，ゴールおよびサブゴールを用いて問題解決をしてゆくことに対して，非常に有益である．問題解決のゴールは，“最高の”レベル，すなわち，$m=1$の問い合せとして記述できる．ゴールに達すると，次に，サブゴールを追求できる．サブゴールに対しては，第2のレベル，すなわち，$m=2$に対して問い合せをする．サブゴールの一つが実現できれば，そのサブゴールに方向づけできる．このようにして，最終のゴールに達したとき，問題が解決される．

要約すると，並列アクセスアルゴリズムは，次のような特徴をもつ．

1. キーワードの和標準形を入力として受けつける．
2. 同じ問い合せに対し，他の記憶セルからレコードを検索する前に，一つの記憶セルからその問い合せを満たすすべてのレコードを検索する．
3. 各々の記憶セルに対し，与えられた問い合せを満たす可能性のあるレコードの最小の集合を検索のために選び出す．
4. 逆ファイルに対しては，レコードを検索する前に，与えられた問い合せを満たすレコードを正確に決める．
5. レコードの検索の重復を防ぐ．
6. 待ち行列に入れたキーワードの和標準形の和形式および階層ファイルへの多重レベルのアクセスを扱うように拡張できる．

このアルゴリズムのこれらの特徴は，可動ヘッドの記憶機器のアクセス機構の動きを最小にすること，二次記憶へのアクセスの数を減少させること，逆ファイルの場合精度の悪いレコードの検索を除去すること，および，データベースに対するオンラインかつ時分割的アクセスを推進することを目指している．

並列アクセスアルゴリズムを利用できない唯一のファイル構造は，索引順序および索引乱ファイルである．これらのファイルのレコードは，単一のキーワードによって特徴づけられているだけであるから，キーワードの積の形式の問い合せはあり得ない．積形式がなければ，主キーワードの選択は必要ない．同じ理由で，レコードが2個以上のリストの共通部分にあるときに，レコードの冗長な検索を除去するという考慮は，不必要になる．これらのファイルの場合，順次アクセスアルゴリズムで十分であろう．

6.5.3　3つの例

並列アクセスアルゴリズムは，次の例によって最もよくわかるであろう．

並列性に対する例

図6.15に示すように，たった1個のセルだけをもつ単純な一般化したファイル構造を考えよう．このファイルは，図6.13に示したフアイルと同じものである．しかし，節点と有向グラフを使って違った形式でこのファイルを説明しよう．レコードを節点で，レコードのリストを実線で示そう．リストの共通部分である節点は，これらの共通部分をもつリストのキーワードを含んでいる．たとえば，09に於けるレコードは，K_1 リスト，K_2 リストおよび K_3 リストの共通部分であり，キーワード K_1, K_2 および K_3 を含んでいる．このファイルには，10個のレコードと，1個の K_1 リスト，1個の K_2 リストおよび2個の K_3 リストがある．ファイルのディレクトリは，次のようになる．

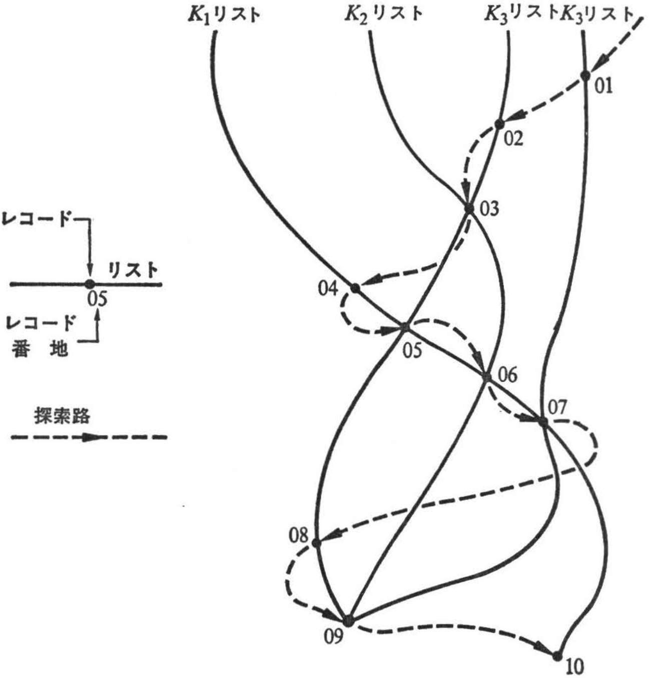


図 6.15　並列アクセス

$\{K_1, 1;\ (00,\ (04,\ 5))\}$
$\{K_2, 1;\ (00,\ (03,\ 3))\}$
$\{K_3, 2;\ (00,\ (01,\ 3)),\ (00,\ (02,\ 5))\}$.

次の形式の問い合せ

$\{K_1 \vee K_2 \vee K_3\}$

に対して，実効的なファイル探索は図 6.15 の破線で示したようになるが，それによれば，アルゴリズムはこれら3個のリストを並列的に探索し，これらのレコードを各々唯一回だけアクセスしている．さらに，最下位の番地から最上位のものへとレコードの走査をしている．

表 6.1　ファイルの探索

a	$Y = g(K, a)$	σ_1	σ
			(01, 02, 03, 04)
01	$07 = g(K_3, 01)$	(07)	
02	$03 = g(K_3, 02)$	(03)	($\underline{01}$, 02, 03, 04, 07)
03	$05 = g(K_3, 03)$ $06 = g(K_2, 03)$	(05, 06)	($\underline{01}$, $\underline{02}$, 03, 04, 07)
04	$05 = g(K_1, 04)$	(05)	($\underline{01}$, $\underline{02}$, $\underline{03}$, 04, 05, 06, 07)
05	$06 = g(K_1, 05)$ $08 = g(K_3, 05)$	(06, 08)	($\underline{01}$, $\underline{02}$, $\underline{03}$, $\underline{04}$, 05, 06 ,07)
06	$07 = g(K_1, 06)$ $09 = g(K_2, 06)$	(07, 09)	($\underline{01}$, $\underline{02}$, $\underline{03}$, $\underline{04}$, $\underline{05}$, 06, 07, 08)
07	$09 = g(K_1, 07)$ $10 = g(K_3, 07)$	(09, 10)	($\underline{01}$, $\underline{02}$, $\underline{03}$, $\underline{04}$, $\underline{05}$, $\underline{06}$, 07, 08, 09)
08	$09 = g(K_3, 08)$	(09)	($\underline{01}$, $\underline{02}$, $\underline{03}$, $\underline{04}$, $\underline{05}$, $\underline{06}$, $\underline{07}$, 08, 09, 10)
09	$0 = g(K_1, 09)$ $0 = g(K_2, 09)$ $0 = g(K_3, 09)$	(0)	($\underline{01}$, $\underline{02}$, $\underline{03}$, $\underline{04}$, $\underline{05}$, $\underline{06}$, $\underline{07}$, $\underline{08}$, 09, 10)
10	$0 = g(K_1, 10)$	(0)	($\underline{01}$, $\underline{02}$, $\underline{03}$, $\underline{04}$, $\underline{05}$, $\underline{06}$, $\underline{07}$, $\underline{08}$, $\underline{09}$, 10)

この記述に対する主キーワードは，K_1, K_2 および K_3 である．ディレクトリのデコードをすると，次に示すような先頭番地の列 σ が得られる．

$$\sigma = (01, 02, 03, 04).$$

ファイルの探索を，**表6.1**に示す．

順次アクセスアルゴリズムも同一のファイルで同一の問い合せに対して適用してあるので，順次および並列アクセスアルゴリズムの相対的な効率を比較することは容易である．この例では，問い合せを満たす10個のレコードを検索するために，並列アルゴリズムは10回のレコードのアクセスをした．一方，順次アクセスアルゴリズムは，16回のレコードのアクセスをして同じ10個のレコードを作り出した．

最小アクセスに対する例

図6.16に示したセル状多重リストを考えよう．そこでは，4個のセルがあり，セルの大きさは10であり，レコードの検索に対し $K_1 \wedge K_2$ の問い合せをする．

ステップ 1． $K_1 \wedge K_2$ の項一つしかないので，最初 K_1 にセル識別名操作子を適用してセル番地の集合 {00, 10, 20, 30} を決定し，次に K_2 に適用してセル番地の集合 {00, 10, 20} を決める．これら2つの集合の共通部分をとると，セル番地の集合 θ {00, 10, 20} を得る．

このステップの意味は，次の通りである．項を満足する任意のレコードは，キーワード K_1 および K_2 の両方を含んでいなければならないので，K_1 および K_2 の両方のリストが存在している記憶セルだけを探索しなければならない．それゆえ，30 におけるセルは，K_2 リストを含んでいないので，探索からはずす．

ステップ 2． セルエントリ操作子を θ に関して K_1 および K_2 に対して適用すると，00 のセルに対し集合 {(K_1, 3), (K_2, 2)}, 10 のセルに対し集合 {(K_1, 2), (K_2, 3)}，および 20 のセルに対し集合 {(K_1, 1), (K_2, 1)} を得る．セル 00 に対しては，K_1 に対するレコード数が 3 であり K_2 に対するレコード数が 2 だから，セル 00 に対する主キーワードとして K_2 を選ぶ．同様の方法で，10 のセルに対して K_1 を主キーワードとして選ぶ．20 のセルに対しては，すべてのレコード数はそのセルに対して同一であるから，主キーワードとして K_1 を任意に選んでいる．したがって，

ディレクトリ

$\{K_1, 4; (00, (06,3)), (10, (12,2)), (20, (23,1)), (30, (35,1))\}$
$\{K_2, 3; (00, (03,2)), (10, (14,3)), (20, (20,1))\}$
$\{K_3, 3; (00, (09,1)), (10, (14,2)), (20, (21,1))\}$
$\{K_4, 3; (10, (16,2)), (20, (22,3)), (30, (37,1))\}$

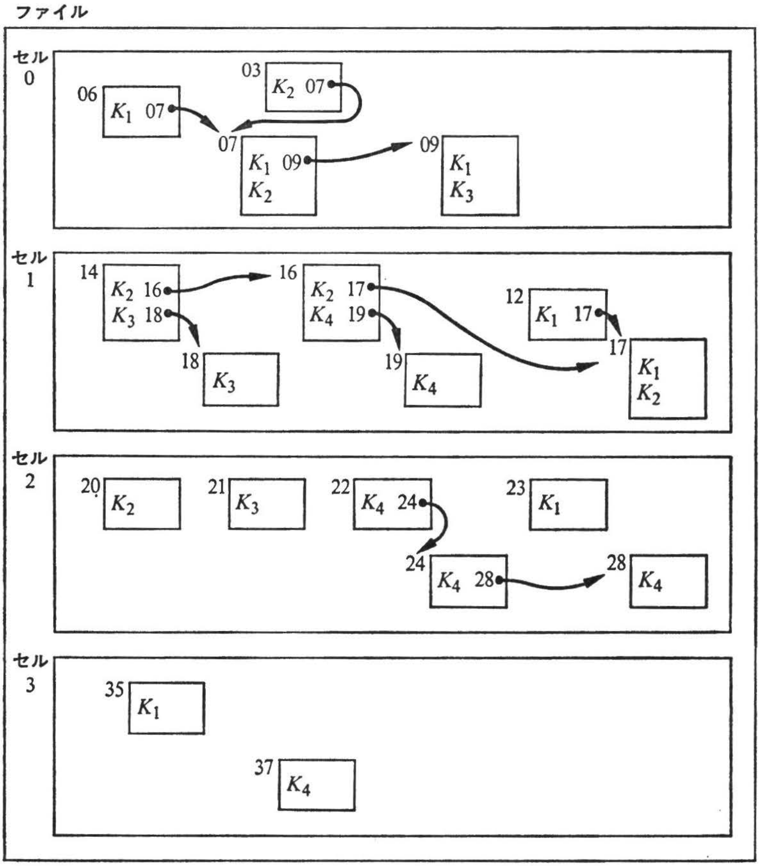


図 6.16　セル状多重リストファイル

$$\gamma = \gamma' = \{(K_2', 00),\ (K_1', 10),\ (K_1', 20)\}.$$

このステップの意味は，次の通りである．一つのセルに対する K リストは，他のセルに対する K リストから完全に分離しているので，一つの項の“主キ

ーワード”は，各セル毎に分離して決めなければならない．与えられたセルの中では，主キーワードは付随しているレコードが最小のキーワードの中から選ぶ．これにより，ファイル探索の間，任意に選び出したキーワードに基づいて探索をする場合に比べて小数のレコードの検索をすれば良いことを保証できる．主キーワードが3個の記憶セルのうちの2個に対して同じものになってしまったけれども，これは一般に必ずしも成り立つものではないことに注意しよう．

ステップ 3.　図6.16のディレクトリを見ると，ディレクトリ探索操作子を$(K_2', 00)$に適用すると番地03が得られ，$(K_1', 10)$に適用すると番地12が得られ，$(K_1', 20)$に適用すると番地23が得られる．したがって，$\Psi = (03, 12, 23)$となる．キーワードは，ここではディレクトリの中でデコードされ，レコードのリストの先頭番地が，ファイル探索処理で使えるようになっている．

ステップ 4., 5., 6. および 7.　これら残りのステップでは，**表6.2**に示すように，レコードの検索を行なう．$\sigma = (03, 12, 23)$が得られている．ファイル

表 6.2

a	$Y = g(K', a)$	σ_1	σ	E
			(03, 12, 23)	
03	$07 = g(K_2, 03)$	(07)		
07	$0 = g(K_2, 07)$	(0)	($\underline{03}$, 07, 12, 23)	
12	$17 = g(K_1, 12)$	(17)	($\underline{03}$, $\underline{07}$, 12, 23)	07
17	$0 = g(K_1, 17)$	(0)	($\underline{03}$, $\underline{07}$, $\underline{12}$, 17, 23)	
23	$0 = g(K_1, 23)$	(0)	($\underline{03}$, $\underline{07}$, $\underline{12}$, $\underline{17}$, 23)	07, 17
			($\underline{03}$, $\underline{07}$, $\underline{12}$, $\underline{17}$, 23)	

探索関数をK_2とσのまだ検索のために使われていない最小の番地とに適用すると，$g(K_2, 03) = 07$となり，これは他のレコードへのポインタである．すなわち，03におけるレコードが検索された．これはK_2しか含んでいないので，このレコードは問い合せ$K_1 \wedge K_2$を満足しない．新しく得た番地は，番地の列σ_1を形成するが，それを，次にσと混ぜあわせる．gに対するパラメータとして使ったσ内の番地を覚えておくために，番地に下線を施す．こうして，

$$\sigma_1 = (07),\ \ \sigma = (\underline{03}, 07, 12, 23)$$

を得る．

次のレコードを検索するために，K_2 と，g に対して使われていない σ 内の最小の番地である 07 とに対して，g を適用する．$g(K_2, 07) = 0$ となり，このレコードは問い合せを満たす．

残りのステップは上と同様であり，表6.2に示してある．ここでは，それについて詳しく論ずることはしない．上記のステップで決めたものと異なった主キーワードを使っていることに注意しよう．

この例では，一般化したファイル構造は，セル状多重リストファイルの構造を正確にモデル化している．このファイル構造とアルゴリズムのもう一つの利点は，逆ファイルに対するディレクトリのデコードの過程を正確にモデル化できることである．これは，下記の例で説明しよう．

絶対精度に対する例

ディレクトリ

$\{K_1, 6; (\ , (01,1)), (\ , (02,1)), (\ , (04,1)), (\ , (06,1)), (\ , (07,1)), (\ , (08,1))\}$
$\{K_2, 3; (\ , (04,1)), (\ , (05,1)), (\ , (07,1))\}$
$\{K_3, 4; (\ , (01,1)), (\ , (02,1)), (\ , (05,1)), (\ , (07,1))\}$

ファイル

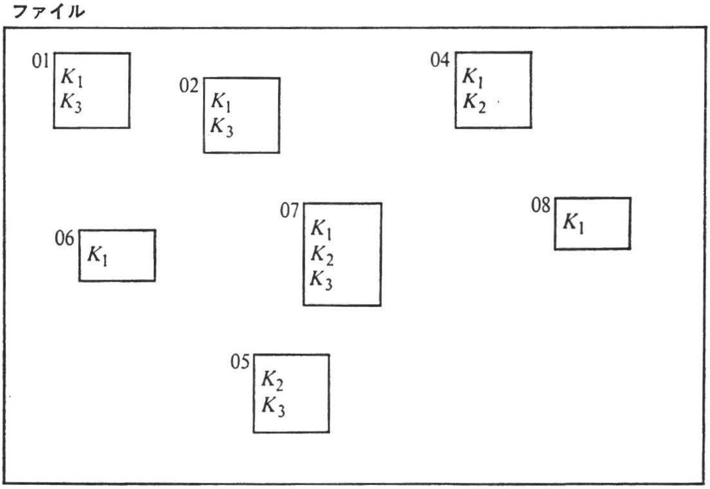


図 6.17　逆 ファ イ ル

図 6.17 の逆ファイルと，問い合せ $K_1 \wedge K_2 \wedge K_3$ を考えよう．

逆ファイル構造において，ディレクトリのデコードの過程では，3つのキーワードに付随した先頭番地の3つの集合の共通部分をとる．このようにして相異なるレコードの番地の集合を選択することにより，ファイル探索の処理は，この3つのキーワードを満足するレコードを正しく検索する．したがって，このアルゴリズムは次のように進む．

ステップ 1.　セル識別名操作子を K_1, K_2 および K_3 に適用して，それぞれ{01, 02, 04, 06, 07, 08}，{04, 05, 07}および{01, 02, 05, 07}の集合を得る．これら3つの集合の共通部分をとると，集合{07}を得る．したがって，セル 07 の中のレコードだけが，問い合せ $K_1 \wedge K_2 \wedge K_3$ を満足する可能性がある．

ステップ 2.　07 におけるセル内で，K_1, K_2 および K_3 に対してセルエントリ操作子を適用すると，集合{(K_1, 1), (K_2, 1), (K_3, 1)}を得る．3個すべてのキーワードに対してレコード数が同じであるから，主キーワードとして K_1 を任意に選ぶ．したがって，

$$\gamma = \gamma' = (K_1, 07)$$

となる．

ステップ 3.　ディレクトリ探索操作子を(K_1, 07)に適用すると，番地 07 が得られる．

従って，

$$\Psi = (07)$$

となる．

ステップ 4., 5., 6. および 7.　07 におけるレコードが検索された．すべてのレコードのポインタは空なので，検索の処理は完了した．

逆ファイルに対してディレクトリ内の先頭番地の集合の共通部分をとることは，一般化したアルゴリズムで，正確にモデル化される．これは，集合の共通部分の処理が，アルゴリズムの主要な要素だからである．

6.6 更　　新

6.6.1 3つの更新アルゴリズム

キーワードを追加したり，削除したり，置き換えたりすると，ファイルを更新しなければならない．

レコード R' に既存のキーワード K_i を追加することは，いくつかの異なった方法で処理してよい．K_i は，既存なので，ファイルのディレクトリには K_i に対するエントリがある．したがって，$(c_{ij}, (a_{ij}, p_{ij}))$ を考慮の対象として選んでよい．その選択の判定基準は，次のように定義される．

R' のレコードの番地を a_{ij}' とし，s を一様なセルの大きさとしよう．

1. 次の式を計算する．

$$c_{ij}' = s\left[\frac{a_{ij}'}{s}\right].$$

ここで，かっこは比の整数部分を示す．

2. 可能ならば，$c_{ij}' = c_{ij}$ となるエントリ $(c_{ij}, (a_{ij}, p_{ij}))$ を選択する．
3. そうでなければ，$c_{ij}' = c_{ij}$，$a_{ij} = 0$，$p_{ij} = 0$ となる新しいエントリ $(c_{ij}, (a_{ij}, p_{ij}))$ を作る．

更新アルゴリズムI.

このアルゴリズムでは，先端からの挿入を行なう．

1. 番地 a_{ij}' からレコード R' を検索する．
2. キーワード K_i を R' に挿入する．
3. $(c_{ij}, (a_{ij}, p_{ij}))$ の a_{ij} を K_i ポインタとして R' に挿入する．
4. R' を a_{ij}' に格納する．
5. ディレクトリ内の $(c_{ij}, (a_{ij}, p_{ij}))$ を $p_{ij}' = p_{ij} + 1$ なる $(c_{ij}, (a_{ij}', p_{ij}'))$ と置き換える．

このアルゴリズムでは，更新されたレコード R' は，先頭番地が a_{ij}' である K_i リストの要素となる．すなわち，R' は K_i リストの先頭である．このアルゴリズムに対する唯一の制限は，番地が c_{ij} であるセルは，ステップ2.や3.で示したようにレコード R' を拡張することができるように十分大きくなければならないことである．

更新アルゴリズムII.

このアルゴリズムでは，置き換えを行なう．

1. 番地の a_{ij}' からレコード R' を検索する．
2. R' の写しを退避する．
3. R' に削除タグを置く．
4. R' を a_{ij}' に格納する．
5. R' の写しに K_i ポインタがもしあれば，取り除く．

6. R' の写しに新しいキーワード K_i を挿入する.
7. R' の写しを新しいレコードとしてファイルに入れる.

削除タグのあるレコードは，その内容がユーザにとってはもはや何の興味もなく，しかも，K_i リストを作るためにはシステムのアクセスアルゴリズムにとって役立つようなレコードである．それゆえ，K_i リストの中の他のレコードにアクセスするために，レコードはそれがたとえユーザに見えなくても，物理的にファイルから削除してしまうことはしない．

ファイル内に新しいレコード R' を置く過程では，$(c_{ij}, (a_{ij}, p_{ij}))$ の a_{ij} を R' 内の K_i への K_i ポインタとして代入する．さらに，これらの $(c_{ij}, (a_{ij}, p_{ij}))$ は $(c_{ij}, (a_{ij}', p_{ij}'))$ と置き換えられる．最後に，R' 内のすべての K_i が K_i ポインタによって代入され，ディレクトリ内の対応する3つ組 $(c_{ij}, (a_{ij}, p_{ij}))$ が処理されると，レコード R' が a_{ij}' に格納される．この処理は，アルゴリズムⅠのステップ3.，4.および5.に似ており，これら3つのステップが R' 内にあらわれるすべての K_i に各々施されなければならないところが異なっている．

アルゴリズムⅡは，アルゴリズムⅠで要求されているように，新しく更新するレコードが既存のセルに適合するように十分小さくなければならないという要求による不利な点はもっていない．更新したレコードは新しいレコードとして入るので，レコードを実現する新しいセルを割り当てることは簡単な仕事である．しかしながら，後ほどタグをつけたレコードを除去することが必要な仕事になる．

更新アルゴリズムⅢ.

このアルゴリズムは，先端からの挿入と置き換えとを結合したものである．これは，前述の2つの更新アルゴリズムの利点を備えている．新しく修飾されたレコードがそれを検索してきたセルと同じところへ入る場合には，いつでも更新アルゴリズムⅠを使う．そうでなければ，アルゴリズムⅡを使う．このアルゴリズムは，タグのついたレコードの数を最小にし，かつ，既存の記憶セルのあふれることも防いでいる．

レコードに新しいキーワードを追加するために，同一のアルゴリズムを使ってもよい．既存のキーワードを追加することと，新しいキーワードを追加することの唯一の違いは，$(c_{ij}, (a_{ij}, p_{ij}))$ の選択にある．新しいキーワード K_i に対するエントリがディレクトリにはないので，次のようなエントリを作成する．

$$\{K_i, 1;\ (c_{i1}, (0, 0))\}.$$

キーワードの削除および置き換えについては，演習問題の一部になっており，ここでは述べない．

6.6.2 構造的な移動の現象

前の議論から，ファイル内に新しいものかあるいは既存のキーワードを追加することは，次の形式の3つ組をディレクトリのエントリとして生成することになることを思い出そう．

$$(c_{ij},\ (a_{ij},\ 1)).$$

すなわち，一つのレコードしか持たない新しい K_i リストが更新のたびに作られるかもしれない．それゆえ，更新の処理により，与えられたファイル構造をリストの短い新しいファイル構造へと変えてゆく作用をもたらす．このような現象は，**構造的な移動**(structural migration)という．

リストごとに唯一つのレコードしかないリストにアクセスするためには，ファイル探索をたくさんする必要はない．それゆえ，新しいファイルの構造は，ファイル探索のオーバーヘッドが低いので，再構成に特に適している．既存のファイルの再構成は，ファイルの更新が頻繁であれば，特に意味がある．6.2.2.2.6節の例で示したように，再構成によりファイルは新しい処理の要求に適合するようにできる．

6.6.3 屑集め系

6.6.3.1 屑集め系の役割

タグのついたレコードに占有されている記憶空間を回復し，リストの中のどのレコードにもタグがついていないようにファイルのリストを再生成することが，**屑集め系**(garbage collector)の仕事である．

セルは，その中のすべてのレコードに削除のためのタグがついているとき，**死んでいる**(dead)という．リストは，その中のすべてのレコードにタグがついているとき，**活動していない**(inactive)という．したがって，死んでいるセル内では，そのセルのすべてのリストは，活動していない．ファイルが生成されて以来どのレコードも削除されていなければ，すべてのリストは活動しており，さらに，すべてのセルは生きている．レコードが削除のためのタグをつけられると，2つの可能性が生ずる．

1. **一つのリストが活動していない状態になるかもしれない．**
   この場合，活動していない K_i リストが占めているセルの一部は，無

駄になる．さらに，K_i に付随しているディレクトリのエントリは，活動していないリストを指しているので，レコードのアクセスのためには何の役にも立たない．セルが死んでいるときは，状況はもっときびしくなる．したがって，活動していないリストを指しているすべてのディレクトリのエントリを除去して，記憶領域の回復をする必要がある．

2. **活動しているリストが，多くのタグ付きのレコードをかかえるかもしれない．**

　リストの中のすべてのレコードにアクセスするためには，タグ付きのレコードを通過する必要がある．それゆえ，タグの付いたレコードによって占有されている記憶セルが無駄になるだけでなく，リストに対するアクセスの間，時間も無駄になる．これは，リスト内のタグ付きのレコードをすべて除去して活動しているリストを“短かく”したいという試みが正当であることを保証してくれよう．

6.6.3.2 屑集めのアルゴリズム

ここで屑集め系のために採用したアルゴリズムは，2つのフェーズから成り立つ．フェーズⅠは，ファイルに対して**唯一回だけ**しか行なわれず，フェーズⅡは必要に応じて**繰り返し**実行される．これらのフェーズを，下に述べる．

フェーズⅠ．

フェーズⅠでは，屑集め系はセル記憶から主記憶へセル内のレコードを出来るだけたくさん読み込んでくる．これは，タグの付いたレコードを無視し，タグのないものだけを処理する．タグのないすべてのレコードに対し，次のことを行なう．

a. レコードのキーワードを抽出する．

b. それらが既にキーワードバッファにあるキーワードと**異なった**ものであれば，それらを**キーワードバッファ**(keyword-buffer)とよぶバッファに入れる．

c. このレコード† に対する新しいレコードの番地を計算する．

d. いまキーワードバッファに入っているキーワードに従って，その番地

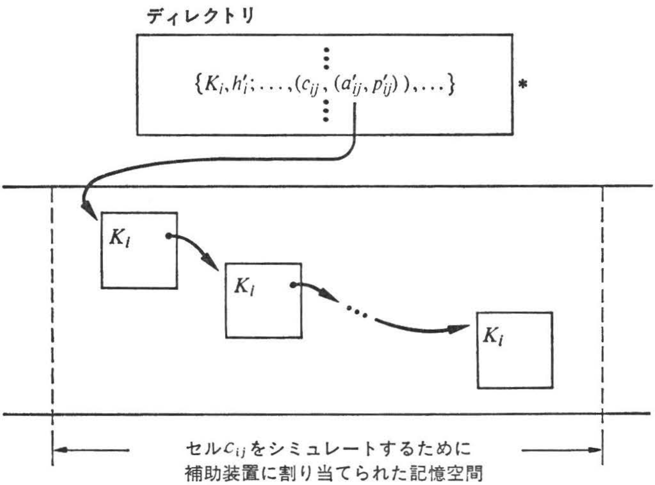
† これは，$|F|$ すなわちファイル空間内の新しいレコードの番地である．これは，一時的記憶としてのみ働く補助記憶装置内の番地とは無関係である．したがって，レコードの番地 a_{ij} を指すときは，ファイルがまだ恒久ファイル空間に写されて**いなくて**も意図するファイル空間内の新しいレコード番地を意味している．しかしながら，システムは，レコードが存在している補助装置上の一時的番地を知っている．

を退避する.

e.　そのレコードを補助記憶装置† に書き出す.

f.　レコードのキーワードバッファ内にあるキーワードと同一であれば，それをキーワードバッファに入れることはしない．その代り，キーワードバッファ内のキーワードに従って退避された番地が，キーワードに沿ってレコード内に挿入される．そして，ステップ **c.** から再開する.

このアルゴリズムでは，与えられた任意の時点で，セル内の最後のレコードの各々のキーワードは，同じキーワードをもつ事前のレコードの番地を含んでいることを保証している．一つのレコードには多数のキーワードがあるので，処理された最後のレコードは，また，リストの最初のレコードである．キーワードバッファ内のキーワードは異なっており，さらに，各キーワードにはリスト内の最初のレコードに対するポインタが与えられていることに注意しよう．このアルゴリズムを，ファイル内のすべてのセルに適用する．キーワードバッファの大きさは，屑集め系が使える主記憶によって制限される．もしも，それがファイル内のすべての異なったキーワードを含むのに充分でない場合は，ファイル処理をフェーズⅡまで続ける．そうでなければ，これはフェーズⅠで終



*$h_i' \leqq h_i$ かつ $p_{ij}' \leqq p_{ij}$ が成り立つことに注意する.

図 6.18　フェーズⅠの終了時

† (前頁脚注参照)

了し，フェーズⅡを開始する必要はない．

どちらの場合も，フェーズⅠの終りには，屑集め系はキーワードバッファ内のキーワードからディレクトリを生成し，ファイルの旧ディレクトリと置き換える．フェーズⅡへ進む必要がないときには，補助装置上に格納してある処理済みのレコードを，セル記憶上に書き戻しておく(図 6.18 参照)．

フェーズⅡ．

キーワードを処理するとき屑集め系はそのキーワードを含む最後に処理されたレコード(したがってリストの先頭)の番地 a_{ij} を退避するだけでなく，このキーワードを含み**このフェーズで**最初に処理されたレコードの番地 a_{ij}^t も退避することを除けば，ここで使うアルゴリズムは，フェーズⅠと同じものである(図 6.19 参照)．恒久的ファイル空間の番地である a_{ij} や a_{ij}' と異なり，a_{ij}^t は補助記憶の番地である．フェーズⅡの終了時には，(主記憶内の)キーワードバッファ内のフェーズⅡで生成された各々のキーワードは，(フェーズⅠで生成された)ディレクトリ内のキーワードに等しいか，あるいは，ディレクトリ内のすべてのキーワードと異なるかどちらかでなければならない．

場合 1

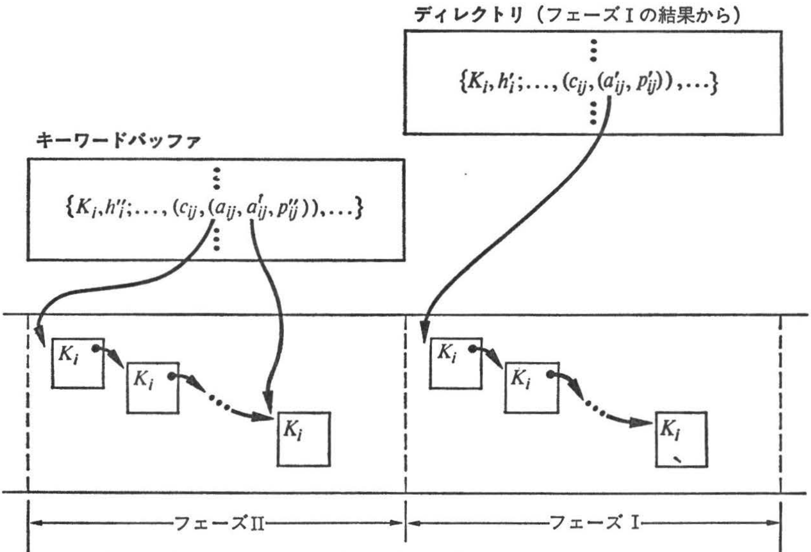


図 6.19　フェーズⅡの各々のキーワードは，2つのポインタをもつ．

キーワードが異なっている場合，屑集め系はそれをディレクトリに入れる．

場合 2

キーワード K_i がディレクトリ内のキーワードと等しければ，屑集め系は，次のようにして結合を行なう．

a）a_{ij}^{t} を使ってこのキーワードをもちフェーズⅡで最初に処理されたレコードを検索し，主記憶に持ってくる．

b）ディレクトリから，このキーワードをもち最後に処理されたレコードのポインタ a_{ij}' を見つける．

c）上記のポインタを，検索してきたレコードに挿入し，そのレコードをもとあったところに戻す(このフェーズで最初に処理されたレコードは，前のフェーズで最後に処理されたものと結合された)．

d）ディレクトリにあるポインタa_{ij}'を，主記憶のキーワードバッファ内の同じキーワードに与えられているポインタ a_{ij} と置き換える．ここで，ディレクトリのキーワードは，このフェーズの最後のレコードを指している．

簡単に言うと，場合2では，図 6.20 に示した破線を作ったことになる．こ

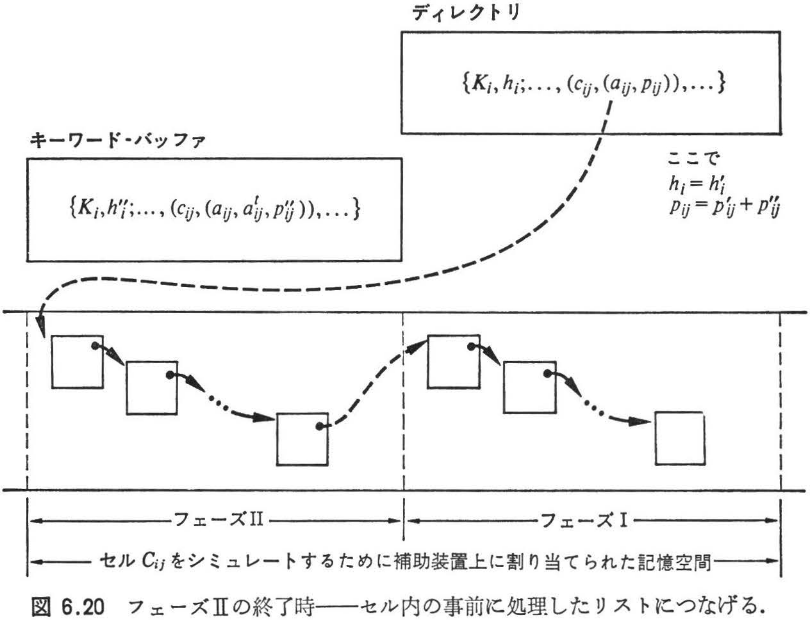


図 6.20　フェーズⅡの終了時——セル内の事前に処理したリストにつなげる．

れは，フェーズⅠの終りでリストの処理が中断した場合に，そのリストを確保することを目指して実行される．

一般に，屑集め系ではフェーズⅠは一回，フェーズⅡは処理するレコードがなくなるまで必要に応じて何回も実行する．処理の終了時には，補助装置上のレコードは，ファイルに割り当てられたセル記憶の中に書き戻されている．

6.6.4 番地の単調性

屑集めのアルゴリズムを開発する際の主な考慮点は，任意のリストにおいてレコードの番地の列の単調性を確保することが中心となる．ファイル内の任意のリストのすべてのレコードの番地が単調に増加(減少)していれば，並列アクセスアルゴリズムでは，直接アクセス記憶機器の可動読み書きヘッドを記憶機器の低位(高位)端から同じ機器の高位(低位)端へ**単一の走査**をするようにできることから，この考慮が必要になる．そうでなければ，リストの単調でない番地列では，読み書きヘッドの再位置付けが生じ時間が無駄になる．したがって，アクセスアルゴリズムを真に有効に使うためには，すべてのリストに対しレコードの番地の単調性を確保しなければならない．

与えられたリストのレコード番地は，ファイルが新しく作成されるときは，いつも単調の順序に並んでいる．更新の際に順序が保たれるならば，アクセスアルゴリズムの有効性も保証されるであろう．このためには，すべてのリストに対し，リスト内のレコードの番地の単調性を確保できるような更新アルゴリズムを開発することを努力すべきである．

最初のファイルが**図 6.21** に示すような例を考えよう．K_j と K_n とを含んでいる図 6.21 のレコード R は，もう一つのキーワード K_i によって特徴づけられていると仮定しよう．すなわち，レコード R は，K_i リストに入れなければ

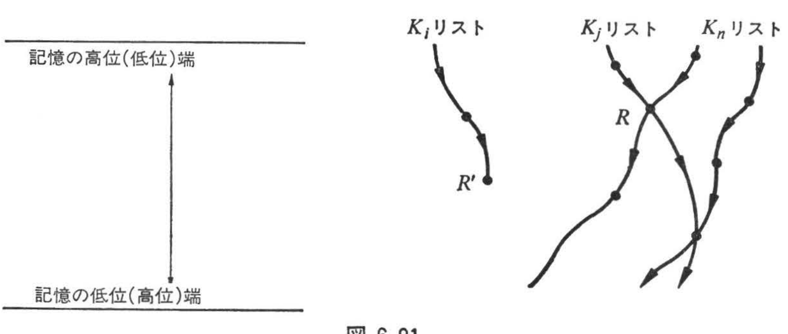


図 6.21

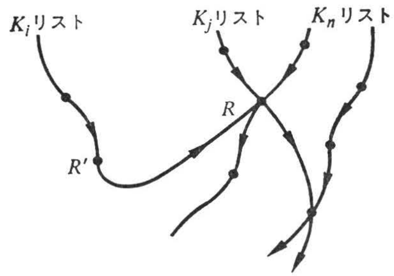


図 6.22

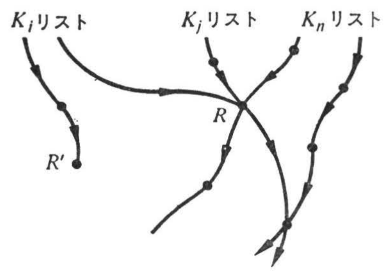


図 6.23

ならない．**図 6.22** に示した更新したファイルは，K_i リストの番地の列が単調減少(増大)の順序になっていないので，受け入れがたい．このようなことが生ずると，アクセスアルゴリズムでは，機器の可動読み書きヘッド機構が，レコード R' を検索してしまった後でレコード R に対して再位置づけされる．一方，**図 6.23** に示したような更新したファイルに対しては，アクセス機構の再位置づけはいらない．この場合，単に，新しい K_i リストを作成するだけである．

6.7　データの機密保護：論理的アクセス制御の研究

データの機密保護は，それぞれの人々がそれぞれの解釈をしている．その理由は，データ機密保護の問題自身にある．それは，手続き，計算機ハードウェア，システムソフトウェア，および，データベースにおける機密保護の考え方を含む大きな問題である．それゆえ，データ機密保護の研究と開発は，これらの問題領域の一つ以上のものを解決することを目指してきた．完全に機密保護をされたシステムのためには，これら4つの領域すべての機密保護の問題を解決することに成功する必要がある．データ機密保護を一般的に理解し，特に，ハードウェア，ソフトウェアおよびデータベースにおける機密保護の役割を理解するために，宝石商の物語から始めよう．

6.7.1　宝石商の問題

宝石商は宝石を売るだけでなく，自分がお客に対して売った宝石をお客のために扱う．修理やサービスをするだけでなく，特に，お得意先で重要なお客の場合，お客の宝石を金庫に預かることを時々しなければならない．宝石商は，宝石の安全性は，金庫の機密保護と同じ程度に過ぎないことを知っている．このために，彼は，精巧な保護機構とその結合とをもった良く作られた金庫を実

現しなければならない．非常にしばしば，宝石をある場所から別の場所へ輸送しなければならない．このような状況では，輸送に対する安全な手続きを工夫しなければならない．輸送手段の中には金庫が含まれているけれども，金庫を運ぶための特別な乗り物(たとえば武装した車)を必要とする．さらに，彼は，宝石を一つの乗り物から別のもの(たとえば，車からボート)へ移すとき，非常に注意深く行なわなければならない．宝石商は，非常に上等な金庫を確保し，非常に信頼性のある輸送の手段や手続きを工夫できるまでは，お客の宝石を適切に守り切ることはできない．

あるとき，宝石商が，宝石を守るためにはそれを隠しておき，宝石の最も重要な役割，すなわち，展示をすることを妨げるようにすることだと理解した場合は，機密保護の問題は複雑化する．そこで，宝石をその権利のある所有者や指定したユーザに対して最大に展示し，同時に，最大の機密保護を提供することができるかどうか彼は考える．宝石商として，腕飾り，首飾りや装身具の中に機密保護の装置を開発できるかどうかを追及するのは自然である．そのような装置が有効であれば，権利のあるユーザによって展示されていても，宝石を保護することができる．この場合，宝石の使用の意図と，その機密保護とが両方とも実現される．

宝石商の店舗の宝石と，データベース・システム内のデータとの間には密接な関連がある．ちょうど宝石を展示しようとするように，データベース・システム内のデータを共用しようと考えている．システム内のユーザがお互に知的に利益を得ることができるのは，データの共用をするからである．しかしながら，データ共用の基礎は，自発性にある．すなわち，システムは，ユーザの私的なデータの匿名性を保護でき，かつ，共用可能なデータのアクセスを制御できなければならない．計算機システムの安全保護の尺度は，いろいろの異なったレベルで見ることができる．宝石を守るために金庫を使う場合のように，データを保護するために実記憶と仮想空間の単位とを使うのは，一つの方法である．このような方法を，**記憶保護**(memory protection)という．一つの場所から別の所へ移さねばならない宝石のように，データも計算機システムの中では動かさねばならない．そのようなデータの移動に対する活動的な執行人は，**手続き**である．これらのものは，主としてプログラムから構成されている．それゆえ，別の方法では，**手続き保護**を開発することを目指す．記憶および手続き保護機構は，両方とも物理的ハードウェアとソフトウェアの要素とを扱うが，こらはまとめて**物理的な保護とアクセス制御機構**と呼んでいる．最後に，データに対し保護機構を実現する試みでは，**論理的な保護とアクセス制御機構**

とよぶもっと精巧な保護機構が必要となる.

6.7.2　アクセス制御と保護の3つのレベル

計算機システムのアクセス制御と保護機構の研究は，一方で私的な情報の匿名性を保護する有効な手段に関するものであり，また，一方では共用可能な情報へのアクセスを制御することに関するものである．アクセス制御の有効的な方法は，ここでは3つのレベル，すなわち，記憶レベル，手続きレベル，および，論理的レベルで考えてよかろう.

記憶のレベルでは，アクセス制御機構は，記憶の単位に基づいて記憶へのアクセスを制御する．主要な点は，保護がその内容ではなく容器に対して適用されることである．その結果，容器の内容物はすべてその周囲と同じアクセス制御に従う．さらに，内容物は，保護された容器にいるときだけは安全である.

物理的記憶保護の機構では，典型的なものとして，記憶領域の限界のアクセスの制御をする記憶限界レジスタあるいは記憶保護“キー”を採用している．ほかにも，もっと精巧な機構も可能である．m 個の記憶領域に対してのアクセス権を監視するための $m \times m$ 行列の制御ビットをもつという考案が進められてきた．たとえば，エントリ A_{ij} は，i 番目の領域から j 番目の領域へのアクセス権を決定する．A_{ij} は，読み出しのみ，読み出し/書き込み，実行のみ，および，特権モードなどのいろいろのアクセス権に対応する.

一般に，一つの領域に対する一人のユーザのアクセス権は，同一の領域に対する別のユーザのアクセス権とは随分違っている．したがって多重プログラミングおよび共用データベースの環境では，システムは，異なったユーザに対して動的に異なるアクセス行列を与えなければならない．それゆえ，仮想空間を利用すると，行列方式の実現を促進することになる．ここでは，ページおよびセグメント表を，命令の解読時にハードウェアが参照する．各ユーザは，自分自身の表を割り当てられ，したがって，これらの表にエントリとして登録されていないような空間のセグメントには，立ち入ることはできない．その結果，アクセス行列のような精巧な機構を，仮想空間を使えば，容易に実現できる．しかし，仮想空間を使っても，保護している領域は，やはり，空間の単位であることに注意しよう.

アクセス制御の第2のレベルは，手続きのアクセス制御と保護である．手続きとは，単純には，プログラムとそれに付随したデータの集合である．それゆえ，記憶保護とは異なり，手続きの保護と制御の概念は，プログラムへのアクセスと保護とに関連している．このために，いつ，どんな条件のもとにプログ

ラムが一つのところから別のところへ制御を渡すのかを決定できる機構を開発しなければならない．換言すると，その機構は，プログラムの実行をその呼び出し，戻り，パラメータの受け渡しなどに関して監視できるものでなければならない．

手続きのアクセス制御と保護の精巧な機構は，“リング機構”として提案され，知られてきた．この同心円的なリング機構により，一つのプログラムはアクセス制御の権利をどれも犯さずに，別のプログラムへ制御を移すことができ，したがって，各プログラムの作業表，データ，中間結果などを保護することができる．概念的には，円心円のリング機構では，ユーザはその手続きを階層的，すなわち，階層の低位部(すなわち，外側のリング)の手続きは特権の少ないアクセス権をもつように並べる必要がある．これは，2つのリングから成る単純な階層の一般化である．2つのリングをもった単純な階層では，手続きは，内側のリング(すなわち，スーパーバイザ状態)か，外側のリング(ユーザプログラムの状態)のどちらかで実行される．スーパーバイザ状態の手続きと通信をするために，ユーザプログラムの状態の手続きは，ゲートまたはゲートの集合を介さなければならない．ゲートの実現は，それぞれの計算機システムによって異なっている．それは，スーパーバイザコール(SVC)として実現してもよいし，ソフトウェアのシステムマクロとしてもよい．一方，ユーザプログラム状態の手続きへアクセスするためには，スーパーバイザ状態の手続きは，ゲートを通る必要はない．リング機構を一般化すると，多くのプログラム実行状態が実現でき，各状態は，同心円のリングで実現される．したがって，たとえば，システムスーパーバイザ状態(一番内側のリング)，ユーザースーパーバイザ状態(次のリング)，ユーザサブシステムモニタ状態，ユーザサブシステム状態，および，ユーザサブシステムプログラムの要素(一番外側のリング)というようにすることも可能である．この機構は，仮想空間をもつかもたないかは別にして計算機で実現できる．それゆえ，仮想空間の保護と手続きの保護は一つでかつ同じものであるという誤った考えに陥入ってはならない．

アクセス制御の最高のレベルは，論理的なものである．大規模なデーターベースの環境では，ユーザは先ず自分のデータをある論理的構造に組み立てる．次に，自分の構造化したデータを，フィールド，配列，レコード，部分ファイル，ファイルといった論理的要素を介して参照する．重要な点は，これらの要素は情報の論理的単位であり，物理的あるいは仮想記憶のイメージとはほとんど相似点がないということである．この論理的単位にアクセス制御の要求と保護の手段とを与えることをユーザに許すことによって，アクセス制御機構は，

その情報がどこにいようとも，情報の直接的制御と保護とを実現することができる．さらに，この機構では，計算機システムの物理的あるいは仮想的記憶の構造をユーザが知っていることを要求してはいない．

したがって，論理アクセス制御機構では，ユーザが共同あるいは私的データをデータベースの論理的要素を使って記述すること，これらの要素にアクセス権や保護の要求を割り当てること，他のユーザが所有してもよいこれらの要素の集合とアクセスの型とを決めること，および，追加の検証あるいはチェックの手段を実現することをユーザが可能になるようにしなければならない．

6.7.3　論理的アクセス制御モデル

ここでは，いくつかの実働データベース管理システムに見られる複雑な論理アクセス制御機構を非常に詳しく論ずる積りはない．そのかわり，単純なモデルを提示し，そこで，これらの機構のいくつかの重要な概念と顕著な特徴とを示そう．

基本的には，論理アクセス制御モデルは，3つの部分から成る．

1. アクセスを制御しなければならない共用データベース．
2. データベースに対するアクセスを制御しなければならないユーザのグループ．
3. ユーザによるデータベースへのアクセスを支配する機構の集合．

このモデルを開発するために，次に述べる問題の解を与えなければならない．これらの解は，別の節で論ずる．

1. データベースの論理的要素を識別する方法．
2. ユーザがデータベースに対して持つアクセスの型の表現．
3. 他のユーザが自分のデータベースに対してもつアクセスの型を割り当てかつ変更できる概念．
4. アクセス可能なデータベースに対してどんなデータ管理の関数を誰が実行できるのかをシステムが決められる基礎．
5. アクセス型で指定されるいろいろのアクセス制御の要求の有効な実現をするための方法論．

6.7.3.1　データベースの論理的要素の識別

データベースの論理的要素は，システムが個々の要素に対し異なった識別とアクセス制御とを決定できるように，一意的に識別されなければならない．し

かし，データベースの個々の要素に一意的な識別名を割り当てる困難さをユーザに押しつけてはならない．したがって，一つの要求は，ユーザではなくシステムがデータベースの個々のかつすべてのデータに対し一意的な識別番号を責任をもって割り当てるようにすることである．ユーザが持っているべきものは，自分の論理的要素をアクセス制御のために記述すること，および，これらの要素に対して自分のアクセス制御の要求を指定することができる手段である．アクセス制御の目的でユーザの論理的データ要素を記述するためには，それらの構造を宣言できなければならない．そして，これらのデータ要素に対するアクセスを事前あるいは事後に制御をするためには，ユーザはそれらの検証と検査の手続きを指定できなければならない．ユーザは，既に，デーベタースの作成の際にデータ構造を宣言するためのデータ記述言語と，データベース管理の要求を指定するためのデータ操作言語とを知っていることに注意しよう．同じデータ記述言語を，アクセス制御の目的のために論理要素の構造を宣言するために使えるかもしれない．データベースの作成および制御の両方に対し同じデータ記述言語を使うと，便利なだけでなく，言語の翻訳におけるあいまいさによって生ずる誤った識別の可能性を防げる．同様にして，同じデータ操作言語は，データベース要素へのアクセス制御に対する検証や検査の手続きを指定するために使ってもよい．それゆえ，第二の要求は，アクセス制御のためにデータベースの要素を記述し，それらに対するアクセス制御の要求を指定するために使う手段は，データ構造を宣言したり，データベース作成や処理におけるデータ操作の要求を指定するために使うものと同じになるべきである．

これらは，モデルが満たそうとすべき要求にほかならない．このモデルでは，データベースの要素は，フィールド(すなわち，属性-値の対)，フィールドの群，キーワード，レコード，サブファイルおよびファイルとして識別する．フィールドに対してはフィールド名(属性)で，フィールド群に対しては群の名前とフィールド間の関係で，レコードに対してはフィールドおよびフィールド名の論理式と算術式を使った選択規準で，サブファイルに対してはキーワードの論理式と算術式で，また，ファイルはファイル名によって参照する．これらの名前と式はすべて，データの定義，操作およびアクセス制御のための手続きの中で使える．これらの使い方については，後節で説明しよう．このモデルでは，手続きは，一方で，データ記述および操作コマンド言語かデータ管理アセンブリ言語のマクロで書くことができ，また，一方で，データ定義，操作およびアクセス制御機能を含むように拡張してきた FORTRAN や COBOL のような高級言語で書くことができる．

6.7.3.2　アクセス型の表現

ユーザがデータベースに対して持っているアクセスの型を，**権限項目**(authority item)として表現する．

概念的には，ユーザと権限項目との間に一対一の対応がある．

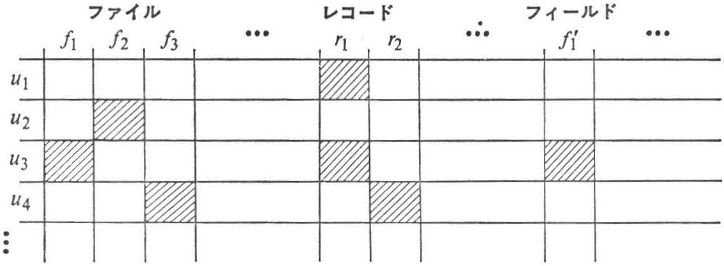
権限項目は，次のものから成り立つ．

1. ユーザの識別名．
2. ユーザがアクセスできるすべてのファイルの名前．
3. ファイルの使用に関する選択(すなわちアクセス権)．これらの選択のいくつかを，下記に示す．

   ND　ディレクトリの読み出しは許されていない．
   RR　レコードだけを読み出す．
   RE　ディレクトリまたはレコードを読み出す．
   EP　プログラムレコードを実行する．
   ER　新しいレコードを挿入する．
   RW　レコードを読み出すか書き込む．
   TB　ファイルが修飾されている間他のユーザからのアクセスを一時的に止める．
   OWN　ファイルを所有する．

4. ファイルレベルの手続き的検証に対するプログラムのエントリ．
5. ファイルの部分をオープンする事に対して，ファイルの部分に対するアクセスを一時的に止めることに対して，またファイルのある部分の使用をある人に対して許したり拒否することに対して記述したレコードへのアクセス制御(すなわち，キーワード，フィールド名およびフィールドの論理式および算術式)．
6. レコードレベルの手続き的検査に対するプログラムのエントリ．
7. フィールドへのアクセス制御の記述(すなわち，フィールドおよびフィールド名の論理式および算術式)．
8. フィールドレベルの手続き的検査に対するプログラムのエントリ．

権限項目をすべて集めたものは，行がユーザで列がデータベースの論理単位となるような**アクセス制御行列**と見做してもよい．エントリ A_{uv} は，ユーザ u が論理単位 v に対して持つアクセス権と制限の列を含んでいる．



実現のためには，この行列はそのままでは疎であり過ぎる．アクセスの権利と制限とを変更し更新することが容易にできるように，この行列を他の(システム)ファイルのレコードと同じように構成すべきである．同じデータ単位に対するアクセスの権利や制限はユーザによって異なっており，ユーザよりもデータ型の方がたくさんあるので，権限項目をレコードとして実現するためには，データ型向きというよりもユーザ向きに考える．すなわち，一人のユーザに対して一つの権限項目を与える．

6.7.3.3 所有権の概念

ユーザがアクセスの権利を認めたり拒否することができるように，所有権の概念を導入しよう．

ユーザは，ファイルの**所有者**(owner)ならば，そのファイルへのアクセスを他のユーザに許可したり拒否したりすることができる．

ファイルの所有者になるためには，ユーザはそれぞれのシステムで異なっている規則や制限の集合を満足しそれに従わなければならない．特にシステム管理者は，事実上の所有者である．彼は権限項目のファイルを所有しているのでシステム内のすべてのファイルを実質的には所有している．したがって，権限項目ファイルへのアクセスがあらかじめ与えられ，しかもファイルの使用は厳重にしなければならないシステム管理者へ課せられる特別な要求がいくつかある．ユーザが一つのファイルの所有者になるときには，システムはそのユーザの権限項目にそのファイルに対する OWN 選択を割り当てる．このモデルでは，ファイルの作成者は，自動的にそのファイルの所有者になる．ファイルの所有者として，ユーザはそのファイルへのアクセスを他のユーザに許可したり拒否したりすることができる．すなわちファイルへの OWN 選択をもつユーザは，そのファイルに対する任意の選択を他のユーザに割り当てることができる．

要約すると，所有権の概念により，システムは図 6.24 に示すようなユーザ

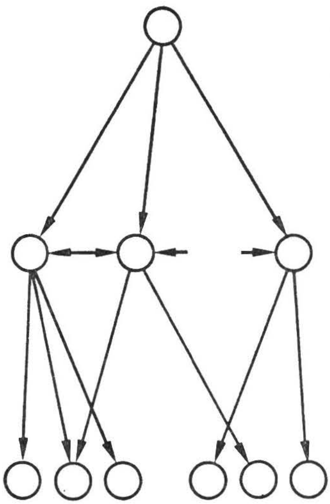


図 6.24　ユーザの階層

の階層を確立する.

6.7.3.4　データ管理コマンドとマクロの使用の基礎

ユーザによるデータ管理のコマンドとマクロの使用は，そのユーザの権限項目の中に割り当てられた選択によって指定される．ユーザの権限項目は，あるコマンドやマクロによって示されている特定のデータベース管理関数をそのユーザが実行できるかどうかを決定するために，データベース管理システムによって常に使われる．たとえば，ファイルの所有者だけがそのファイルに対するアクセスを他のユーザに与えることができる．それゆえ，次のコマンドを実行するためには，コマンドのユーザは所有者でなければならない．すなわち，そのユーザの権限項目の中に OWN 選択がなければならない．

XFILES ACCESS/(ユーザ名)/(ユーザのプロジェクト名)/(ファイル名)/(選択)/(保護の記述)

XFILES ACCESS コマンドにより，ファイルの所有者はそのファイルの使用

のために他のユーザに対してアクセス権を派遣することができる．それは，基本的には次に示す5個のパラメータを取る．

1. ユーザ名：　これは名前または名前の論理和でよい．
2. ユーザのプロジェクト名：　単一のプロジェクト名．
3. ファイル名：　これはファイル名またはファイル名の論理和でよい．
4. 選択：（前節の選択のリストを参照）．
5. 保護の記述：　アクセスへの恒久的な保護に関して，サブファイルを特徴づけるキーワードの論理式あるいは算術式．

```
XFILES ACCESS/BROWN/OSU/
                FILE1/RR/(INFORMATION TYPE, PRIVATE)/
                FILE2/RR/(CLASSIFICATION, CONFIDENTIAL) ∨
                         (PRIORITY, IMPORTANT)
```

上記の例は，Brown 氏が FILE 1 および FILE 2 の両方にアクセスしてもよいことを示している．彼は，FILE 1 の(INFORMATION TYPE, PRIVATE)と FILE 2 の(CLASSIFICATION, CONFIDENTIAL) ∨ (PRIORITY, IMPORTANT)という保護の記述を満足するもの以外は，すべてのレコードを読むことができる．

一方，ファイルの所有者でなくてもファイルに対してユーザが実行できるコマンドがある．

XFILES　OPEN
XFILES　OPEN/A_1/B_1/A_2/B_2/$\cdots$/A_n/B_n
XFILES　OPEN/A_1/B_1/A_2/B_2/$\cdots$/A_n

最初のファイルのオープン実行コマンドは，ユーザがアクセスできるすべてのファイルをオープンする．他の2つの形式は，ユーザがアクセス可能な限り，論理式および算術式 B_i を満足する A_i の部分(すなわちサブファイル)をオープンする．ある i に対して B_i が空であったり，それが抜けていたりする場合は，ファイル A_i は全部オープンされる．

しかし，A_i は単一のファイル名かあるいはファイル名の論理和である．たとえば，役職名が一級の課長であるような従業員レコードすべてから成り立っているサブファイルを処理するためにオープンしたかったら，次のものを実行すればよい．

```
XFILES OPEN/EMPLOYEE-FILE/(JOB-TITLE, FOREMAN-FIRST-CLASS)
```

コマンドを使用するためにユーザの権限項目の中からシステムに与えるべき選択（アクセス権）	いくつかのコマンド（とその説明）
な し	NEW FILE（新しいファイルを作成）
OWN	DELETE FILE（既存のファイルを消去）
RR, RE, RW, OWN, EP, または ER	OPEN SUBFILE または FILE(アクセスするためにサブファイルまたはファイルをオープン)
な し	CLOSE FILE（ファイルをクローズ）
OWN	ACCESS SUBFILE また FILE(他のユーザに自分のファイルあるいはサブファイルへのアクセス権を与える)
OWN	NO-ACCESS SUBFILE または FILE(自分のサブファイルあるいはファイルを与えたアクセス権をとり戻す)
TB または OWN	TEMPORARILY BLOCK SUBFILE または FILE（サブファイルあるいはファイルへの排他的アクセス要求）
な し	UNBLOCK FILE
RW または OWN	RECOGNIZE FILE（屑を集める）
な し	LOG-IN FOR A FILE(ファイルレベルの手続きに対しオープン時に必要なパラメータを挿入)
OWN	ENTER A LOG-IN PROGRAM(ファイルに対するファイルレベルの認証手続きをもってくる)

図 6.25 データ管理コマンド

ここで，EMPLOYEE-FILE は，ファイル名であり，(JOB-TITLE, FOREMAN-FIRST-CLASS)はキーワードである．

図 6.25 には，ファイルとサブファイルに対するコマンドをそれを実行するときの付随した選択と共に表にしてある．

システムマクロは，コマンドと同じような制御の選択に従う．これらのマクロは，データ管理の関数としてユーザのアセンブリプログラムの中で使われる．コンパイラを介すると，ユーザはこれらのマクロをデータ管理副プログラムとサブルーチンへの呼び出しとして使うこともできる．

マクロ呼び出しは，次のような形式をとる．

GSR $P, (S_1, S_2, \cdots, S_n)$

ここで，GSR はデータ管理システムのマクロ名，S_i はサービスの名前，また，P は空か，キーワードの論理式あるいは算術式の記号番地，または番号である．

たとえば，ユーザプログラムがオープンしたファイルに対して新しいレコードを挿入したいときには，ユーザは次のようなデータ管理のサービスを有するマクロ呼び出しをすることができる．

サービス名	**意　味**
IT, FQ	新しいレコードを指定したファイルに挿入する．

マクロ呼び出し

GSR　, (IT, FQ)

レコード IT の番地をレジスタ i に，ファイル名 FQ をレジスタ j に入れておく．

呼び出しからの戻り

1. サービスが成功裏に完了したら，レジスタ i には零が入っている．
2. そうでなければ，レジスタ j はすべて 1 で p ビットが立っている．

この例では，パラメータ P は空である．次の例では，パラメータ P を論理式および算術式へのポインタあるいは番号として使っている．

サービス名の結合	**意　味**
DL, IM, FQ	指定したファイルから記述に従いレコードを消去する．

マクロ呼び出し

GSR　ADDR, (DL, IM, FQ)

ここで記述の番地は ADDR であり，ファイル名はレジスタ FQ にある．

次のマクロにより，プログラムは実行コマンドのオペランドとしてパラメータを持ってくることができる．

サービス名	**意　味**
DES	現在のコマンドの指定したオペランドの番地を取り出す．

マクロ呼び出し

GSR　N, (DES)

ここで N は N 番目のオペランドであることを示す．

データ管理に対する上記のサービス用マクロが使えるというだけでは，システムのユーザにこのようなデータ管理のサービスを受けられる権利を与えたことにはならない．実際，ユーザの権限項目の中に ER 選択がなければ，ユーザは最初のマクロ(すなわち，IT, FQ サービス名をもつもの)の実行はできない，ファイルの所有者でなければ，ユーザは第2のマクロ(すなわち，DL, IM および FQ サービス名をもつもの)を出して，ファイル内のレコードを除去することはできない．マクロの DES を使うのには，何の選択もいらない．その理由は，ユーザのプログラムは，自分のコマンドのパラメータをいつも知ることができるからである．実際，これは，実行中のプログラムが処理中のコマンドと連絡をとる有効な方法である．まとめとして，**図 6.26** にデータ管理のマクロとそれに必要な選択との集合を表にしてある．

6.7.3.5　機密保護の手続き

ユーザが望んでいるアクセスの型とは無関係に**機密保護の手続き**があり，それを介してデータベースへのすべてのアクセスを検証し監視しなければならない．

1. コマンドまたはマクロ呼び出しの識別をする．
2. 関係するファイルに対してユーザがアクセスできるかどうかを調べる．すなわち，その権限項目を見る．
3. ユーザがファイルに対してアクセスできるならば，そのファイルが使えるように現在オープンされているかどうかを調べる．
4. ファイルがオープンされていれば，次に，呼び出しに関して適切なファイルの選択(たとえば，検索に対し読み出しのみの選択，入力に対し書き込みの選択など)をそのユーザが持っているかどうか調べる．
5. ユーザがそのファイルの使用に対して正しい選択をもっているならば，関係するシステムプログラムに対して必要な情報を設定する．
6. 式の事前の解析を行なう．式の中のキーワードや属性-値の対が禁止されていれば，それらのキーワードや対を取り除く．
7. 要求されたサービスを実行するために，適切なシステムプログラムを呼び出す．
8. サービスの状況を追跡する．データ管理のサービスは繰り返し呼び出さないと完了しないかもしれないので，後でサービスを継続するために，いくらかの情報を退避しておく必要がある．
9. 最初にシステムプログラムに対して設定しておいた情報を更新し退避

マクロ使用の許可を得るために必要な省略形	いくつかのデータ管理マクロ GSR $P_i\ (S_1, S_2, \cdots, S_n)$	
	マクロ呼び出し	意味
RE	GSR ,(KY)	すべてのオープンしたファイルのディレクトリから，キーワードを持ってくる。
	GSR ,(KY,FQ)	mq レジスタ内に名前がある指定したファイルのディレクトリから，キーワードを持ってくる。
	GSR A,(DL,IM,D)	Aにあるキーワードの論理式によるだけで，データレコードを消去する。
	GSR A,(DL,IM,MP,ND,NMU,NSYS)	Aにある式だけで，プログラムレコードを消去する。
OWN	GSR A,(DL,IM,MU,ND,NMP,NSYS)	式によるだけで，コマンド手続きレコードを消去する。
	⋮	⋮
	GSR A,(DL,IM,SYS,ND,NMP,NMU)	式によるだけでシステムレコードを消去する。
	GSR A,(DL,IM,D,MP,NMU,NSYS)	式によりデータとプログラムのレコードを消去する。
（なし）	GSR N,(DES)	実行中のコマンドのN番目のパラメータの番地をアキュムレータにとり込む。
RR	GSR A,(NDL,IM)	Aにある論理式によりレコードを検索する。
	GSR A,(NDL,IM,FQ)	指定したファイルから式によりレコードを検索する。
ER	GSR ,(IT)	レコード番地がアキュムレータにある新しいレコードをオープンしたファイルへ挿入する。
	GSR ,(IT,FQ)	新しいレコードを指定したファイルへ挿入する。

図 6.26 いくつかのデータ管理マクロ

する．

10. ステップ3.において一個以上のオープンされたファイルが関係しているならば，オープンされた次のファイルのサービスを続ける．
11. 各々の出力レコードがファイルのオープンした部分に属していること，他から使うことを一時的に止めていないこと，アクセスから恒久的に保護していないことを確かめる．
12. レコードレベルでの手続き的検査を満足する．
13. アクセスから保護されているフィールドが出力レコードから除かれていることを確かめる．すなわち，事後の制御である．
14. フィールドレベルでの手続き的検査を満足する．

機密保護の手続きのいくつかのステップについて論じよう．

ステップ1.と4.とでは，特定のコマンドやマクロ呼び出しを発するために必要な選択の型をシステムが識別することが必要である．前の節に示したように，ユーザの実行できるデータ管理のコマンドやマクロは，彼に割り当てられた選択の型によって決まる．たとえば，OWN 選択がなければ，ユーザは(ファイル内のすべてのレコードを示す論理式と一緒に)データ管理コマンド DELETE FILE か マクロ呼び出し IM-DL-FQ を発することによってファイルを消すことは許されない．このために，システムはユーザの権限の項目を参照する．ファイルに関する選択は権限項目の第3のエントリの一部として常にファイル名に対して与えられているから，システムはユーザがそのコマンドやマクロ呼び出しを使うために適切な選択を有しているかどうか決定することができる．

ステップ2.に対しては，システムは権限項目の第2エントリを参照する．このエントリにはそのユーザに対してアクセス可能なすべてのファイルの名前の一覧表があることに注意しよう．しかし，そのユーザがアクセスできないファイルは，その権限項目に名前が入っていない．すなわち，アクセス不可能なファイルは，ユーザにとっては完全に見えないのである．

機密保護の手続きにステップ3.が含まれているのは，ファイルの所有者がそのファイルのすべてのユーザをふるいにかけるために認証プログラムを使う最初の機会を持てるようにするためである．ファイルに対する認証プログラムの典型的なものは，ファイルの作成時に ENTER-A-LOG-IN PROGRAM コマンドを使ってファイルの所有者がシステムにロードしてくる所有者の書いたルーチンの集合からできている．ユーザがファイルをオープンしようとすると，そのファイルに対する認証プログラムをユーザの意志と無関係に呼び出す．プログラムはそのユーザから[GSR n,(DES)マクロを介して]いろいろの入力を要求

するかもしれないが，それは肯定的あるいは否定的な認証を示す標準的な2つの出力のうちのどちらか一つを常に生成する．認証のために書かれたプログラムはデータ管理マクロを使うことができるので，ファイルのオープンを試みた数とユーザのパスワードの結合とに関する情報を格納したり検索したりすることができる．肯定的な認証がなければ，ファイルはアクセスするためにオープンされることもないであろう．

ファイルレベルの検査に対するプログラムのエントリポイントは，権限項目の第4のエントリに入っていることに注意しよう．それゆえ，認証プログラム(あるいはログイン手続きともいう)をファイルレベルで注意深く使うと，全体としてファイルの保護を非常にうまく行える．

ステップ**5.** とステップ**7.** から**10.** までは，自明である．ここでは，詳しく説明はしない．ステップ**6.** では，このユーザによって使うことが禁止されているキーワードを取り除く．禁止されているキーワードに関する記述は，ユーザの権限項目の第5エントリに置かれている．ユーザの問い合せ，マクロの記述，およびコマンドの記述からキーワードを除くことにより，このキーワードに基づいたアクセスは決して生じない．それゆえ，ユーザは，検索が成功あるいは不成功したことを見ただけで，キーワードとそれに付随したレコードが存在するかあるいは存在しないかを推定することはできないであろう．ステップ**11.** では，機密保護の手続きは，ステップ**3.** のようなファイルレベルのアクセス制御からサブファイルレベルのアクセス制御へ移る．キーワード，フィールドおよびフィールド名の論理式や算術式をファイルからサブファイルへ分割する手段として使うと，システムはサブファイルへのアクセスの制御をすることができる．しかしながら，この場合，分割は仮想的に行なわれ，サブファイルは実際には何も生成されていない．物理的にサブファイルを複写しない理由は，一方でデータベースの保全性を保護することであり，もう一方で，更新をしたいからである．仮想サブファイルは，新しい論理および算術式を導入することによって，すぐに作成できる．実際，**図6.27**に示すように，いろいろのアクセスのためにサブファイルの重ね合せがあり得る．

知る必要があるということに基づいて，多くのサブファイルが**表6.3**に示すように同一のファイルに対して定義されるであろう．いろいろのアクセス制御の要求に対するサブフイァルの多重度は，大きくなるかもしれない．しかしながら，レコードがサブファイルに属するかどうかを検証するためにシステムにとって必要な機構は，すぐにわかる(**図6.28**参照)．

アクセス可能なレコードとは，サブファイルレベルですべての機密保護の検

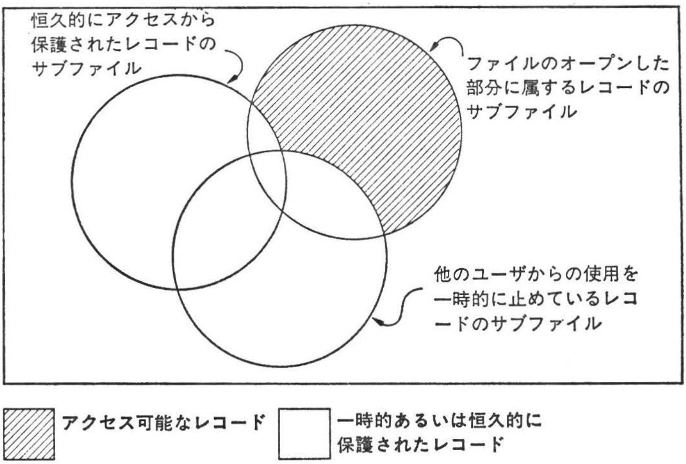


図 6.27　サブファイルの保護

査で検証されてきたレコードのことである．これらの検証は，図6.28に示すように容易に機械化できる．

ステップ12. では，システムは，ユーザにアクセス可能となったレコードの検査をする手続きをファイルの所有者が実現できるようにする．ステップ3.

表 6.3　いろいろのアクセス制御のためにサブファイルを定義する論理式および算術式の使い方

サブファイルレベルのアクセス制御に対する意図した式	レコードは式を満たすか？	レコードのアクセス可能性
使用からの恒久的保護	は い	拒 否
	いいえ	承 認
使用のための一時的オープン	は い	承 認
	いいえ	拒 否
他のユーザから使用を一時的に停止	は い	拒 否
	いいえ	承 認
⋮	⋮	⋮

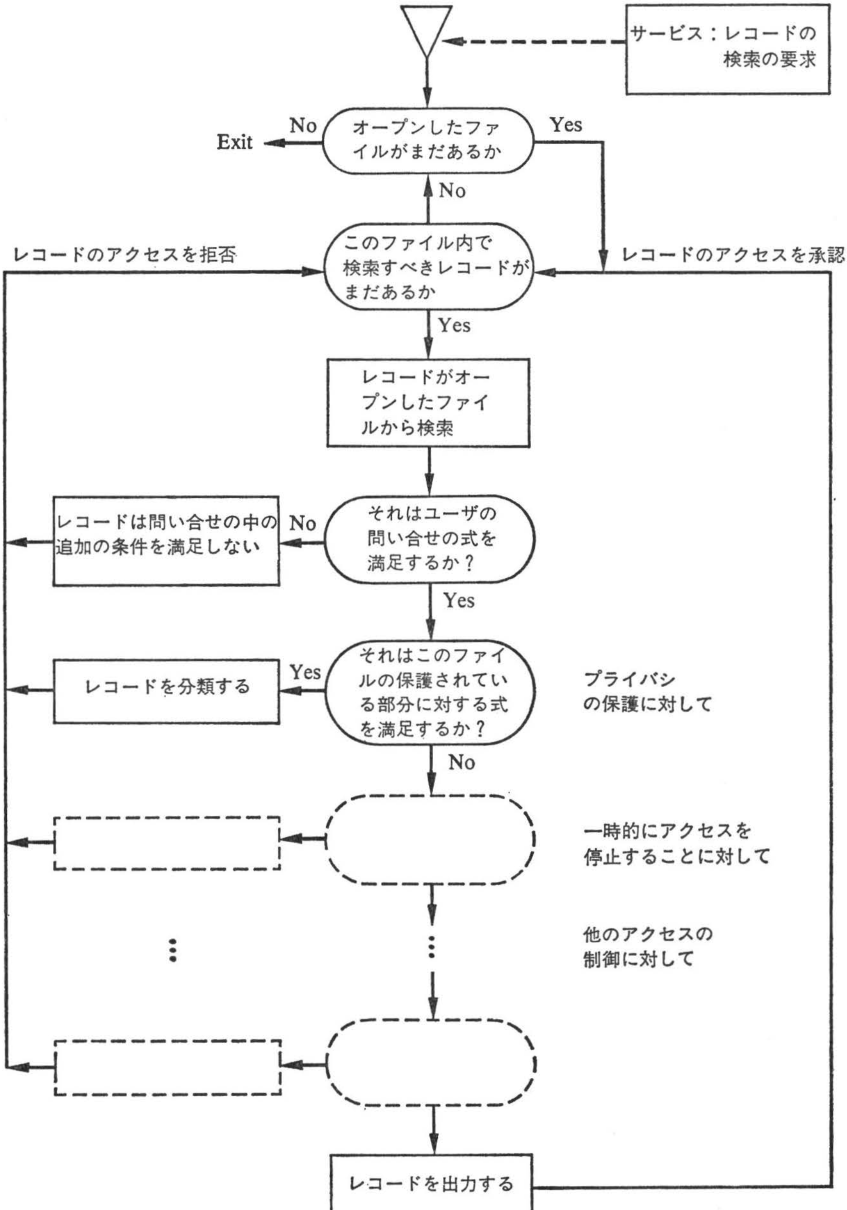


図 6.28 レコードアクセスにおける式の検証

ではアクセス制御はファイルレベルであり，ステップ11. ではアクセス制御はサブファイルレベルであるのに対し，このステップではレコードレベルのアクセス制御を可能とする．ファイルの所有者が自分自身のレコード検査手続きを開発できるようにして，他のユーザに既にアクセス可能となっているレコードを更に検査することができるようにしている．

ステップ14. では，システムは保護されたフィールド名の事後の検査を実行する．この検査は，レコードが二次記憶からシステムの作業領域へ検索されてきた後でだけ可能となる．事後の検査とは異なり，(ステップ6. における)問い合せやユーザのプログラム実行時間に関するフィールド名やフィールドの事前の検査では，すべての保護されたキーワード，フィールド名，および，フィールドをユーザのデータ管理とマクロ呼び出しとから取り除いている．このようにして，“不正な”キーワードによるアクセスは何も始まらない．ステップ14. は，機密保護の手続きの最後のステップである．アクセス制御が最終的にユーザのフィールドレベルに達するのは，このステップである．多くのアクセス制御の要求は，フィールド名および(あるいは)値の何らかの結合に基づいているから，名前の整合と値の計算とは，動的に行なわなければならない．フィールドレベルにおいてこれらの整合と計算の手続きを実現することにより，ファイルの所有者は他のユーザが自分のデータにアクセスすることを直接制御することができる．

機密保護手続きは，データベース管理システムの主要な部分である．

6.7.4　データの機密保護に関する要約

われわれは，単純な論理アクセス制御モデルを論じてきた．このモデルの特徴は，次の通りである．

1. データベースのアクセスおよびアクセス制御の両方に対する論理的参照——物理記憶および仮想空間構成に対する参照は行なわない．
2. アクセス制御のために情報をまとめるための柔軟な区分化の技法——このまとまりは，ファイル，サブファイル，レコードおよびフィールドである．ファイル，レコードおよびフィールドを区分化するための手段として論理式および算術式を使うので，物理的にサブファイルを生成する必要はなくなる．さらに，式は権限項目に格納され，区分化したデータからは離される．これにより，式を更新するだけで区分を柔軟に変えることができる．
3. 検査と認証をさらに行うための手続き的方法——各々の論理レベル

(ファイル，サブファイル，レコードあるいはフィールドレベル)においては，ファイルの所有者は，自分のデータベースを他人が使うことを検査，監視，あるいはふるいにかけるために，自分自身のプログラムを導入することができる.

4. モジュール的機密保護手続き——手続き的検査と区分化の手段のユーザによる使用は，変り得る．一方で，公共的ファイルに対しては，ファイルの内部には区分化はあり得ず，どのレベルでも手続き的検査もあり得ない．また，一方では，きちんと区分化したファイルに対する多重レベルの検査を，ファイルの所有者が要求できる．機密保護の手続きはモジュール化しているので，ユーザの機密保護の要求を支援するためにちょうど良い程度の機密保護の検査を系統的に実行することができる.

5. 同一のデータの集合に対する多重アクセスを停止したり停止を解いたりするための相互ロックの機構——この機構は，たとえば，一人のユーザがあるデータの集合を更新しようとし，別のユーザがその更新の終了する前に同一のデータの集合に対してアクセスしようとするようなオンラインの環境で必要になる.

6. フィールドレベルのアクセスに対する事前および事後の制御——事前の制御は，問い合せの文のアセンブリ時と，コマンドやマクロの実行時とに行なわれる．事後の制御は，データの集合がユーザに対して出力されようとするときに実施される．前者は，不当な問い合せや誤ったコマンドや要求によってアクセスが生ずることを防ごうとしている．後者は，システムの出力から高度に分類した性質の小さなデータを取り除こうとして設計してある.

7. 使用の容易性——所有権の概念を導入したので，システムのユーザはアクセスの権利や制限を得たり拒否したりする順序だった方法を使える．アクセスの制御情報は権限項目の中にシステムファイルのレコードとして格納されているので，これはシステム管理者によって非常に容易に更新できる．さらに，アクセス制御情報を生のデータから分離させたので，生のデータベースを処理しなくても，ユーザのアクセスの要求を変更することができる.

6.8 あ と が き

一般化したレコードの構成は，[8] から取った．レコードの型紙の実現は，[5] で述べている．一般化したファイル構造とその並列アクセスアルゴリズム

とは，[11] および [13] に基づいている．屑集め系は，[7] から取った．商業用データベース管理システムのいくつかの概括に関しては，読者は [1] を参照するとよい．索引順序ファイル [15]，逆ファイル [22]，多重リスト [21]，および，セル状多重リスト [18] のような良く知られたファイル構造は，実際の世界での応用に使われてきた．並列アクセスアルゴリズムを改良しようとする試みは，[20] にある．キーワードを一般化して，属性-値の対の最小項を表わし，最小項を使ってディレクトリ内のリストのポインタやレコードの番地を決めることについては，読者は [23] を参照するとよい．

データの機密保護システムに対するアクセス制御の機構は，[6] および [9] の仕事に基づいてモデル化したものである．[4]で述べているように，最初の実働システムは，[7] で紹介されている．アクセス制御をサブファイルのレベルからデータのレベルへ拡張したのは，[2] で推進された．医療関係の応用で微妙な情報を保護するためにデータ機密システムを使ったのは，[2]と [14] に見られる．行列方式を使った記憶保護の文献は [17] にあり，リングの概念を使ったプロセスの保護は [3] によって推進され，機密保護のデッドロック機構を使った論理的保護，文脈的保護，および一貫性の制御に関しては，[10] に概略が述べてある．

参　考　文　献

1. CODASYL Systems Committee. *A Survey of Generalized Data Base Management Systems*, ACM, New York, May 1969.
2. M. Gelblat. "Computers for Medical Problem Solving—A New Medical Environment." Ph.D. Dissertation, The Moore School of Electrical Engineering, University of Pennsylvania, December 1971.
3. R. M. Graham. "Protection in an Information Processing Utility." *Comm. ACM* **11,** 5 (May 1968): 365–369.
4. L. J. Hoffman. "The Formulary Model for Access Control and Privacy in Computer Systems." Ph.D. Dissertation, Stanford University, May 1970.
5. M. Horton. "Reading, Writing, Creating and Updating Records and Files in a Generalized File Structure." Master's Thesis, University of Pennsylvania, 1971.
6. D. K. Hsiao. "Access Control in an On-Line File System." *File Organization-Selected Papers from File 68*, IFIP ADP Group, Amsterdam: Swets and Zeitlinger, 1969.
7. D. K. Hsiao. "A File System for a Problem Solving Facility." Ph.D. Dissertation, University of Pennsylvania, May 1968, NTIS AD 671 826.
8. D. K. Hsiao. "A Generalized Record Organization." *IEEE Transactions on Computers* **C20,** 12 (December 1971): 1490–1495.

9. D. K. Hsiao. "Logical Access Control Mechanisms In Computer Systems." *Proceedings of ADP Secure Data Sharing Conference*, Washington, D.C., September 1972. Also available as a technical report (OSU-CISRC-TR-73-4), The Ohio State University, Columbus, Ohio.

10. D. K. Hsiao et al. "Research on Data Secure Systems." *Proceedings of the AFIPS 1974 NCC* 43, AFIPS Press, Montvale, New Jersey, May 1974.

11. D. K. Hsiao and F. Harary. "A Formal System for Information Retrieval from Files." *Comm. ACM* **13,** 2 (February 1970): 67–73; Corrigenda. *Comm. ACM* **13,** 4 (April 1970): 266.

12. D. K. Hsiao and F. Manola. "A Uniformed Approach to Structure, Access and Update in Data Base Systems." *Proceedings of European Computer Congress 74*, London, England, May 1974.

13. D. K. Hsiao and F. Manola. "Data Management with Variable Structure and Rapid Access." *Proc. of First USA-Japan Computer Conference*, Tokyo, Japan, October 1972.

14. D. K. Hsiao and K. Nakanishi. "A Cardiac Catheterization Information System—An Application of an Advanced Data Management Facility." *Proc. of ISLANDS OF APPLICATIONS—Tokyo Conference*, IEEE Computer Society, Tokyo, Japan, June 1972.

15. "IBM System/360 Operating System: Data Management," Form C28-6537-2, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y. (November 1966).

16. D. E. Knuth. *The Art of Computer Programming, Vol. 1, Fundamental Algorithms.* Reading, Mass.: Addison-Wesley, 1968, pp. 316–329.

17. B. W. Lampson. "Scheduling and Protection in an Interactive Multi-Processor System." Ph.D. Dissertation, University of California at Berkeley, March 1967.

18. D. Lefkovitz and R. V. Powers. "A List-Structured Chemical Information Retrieval System." *Proceedings 3rd National Colloquium on Information Retrieval*, Philadelphia, Pennsylvania, May 12–13, 1966.

19. F. A. Manola. "An Extended Data Management Facility for a General-Purpose Time-sharing System," Master's Thesis, May 1971, NTIS AD-724 801.

20. J. Pomeranz. "Fewer Retrievals in the Hsiao-Harary File Structure." *ICR Quarterly Report No. 27*, The Institute for Computer-Research, University of Chicago, November 1, 1970, II D-1 to II D-4.

21. N. S. Prywes. "Man-Computer Problem Solving with Multilist." *Proc. IEEE* **54,** 12 (December 1966): 1788–1801.

22. A. H. Vorhaus and R. D. Wills. "The Time-Shared Data Management System: A New Approach to Data Management." System Development Corp., Santa Monica, California (February 1967) (SP-2747).

23. E. Wong and T. C. Chiang. "Canonical Structure in Attribute Based File Organization." *Comm. ACM* **14,** 9 (September 1971): 593–597.

演習問題

1.　レコードの仕様を与える簡単なデータ定義言語を設計せよ．データ定義言語を入力として受け取り，指定された構成のレコードの型紙を出力として出すような**レコード入力プログラム**を作成せよ．さらに，レコード入力プログラムは，新しい論理レコードを入力として受け付け，型紙の情報に基づいてレコードを検証し，正しければ個々のレコードに対し適切なレコード制御ブロックを生成し，出力としてレコード制御ブロックとそれに付随したレコードの記憶イメージを出さなければならない．

2.　1番目の問題で生成したレコード制御ブロックとレコードの記憶イメージとに基づき，新しいレコード構成を反映するようなレコードの型紙の追加を定義せよ．新しい型紙と既存のレコードイメージと制御ブロックを入力として受け取り，論理レコードを出力として出すような**レコード出力プログラム**を作成せよ．

3.　(W. Kuhn)　レコードの構成を，根のある順序づけられたラベル付き木と見なすと，与えられたレコード構成に対する可能な構造の数を決定することができる．更に詳しくいうと，与えられたn個の属性をもつレコード構成に対して，(1) kまたはそれ以下の属性をもつ場合，(2) ちょうどk個の属性をもつ場合，に対して，可能な構造の数を計算する再帰方程式の集合を作り出せ．

4.　セル状多重リストのファイル構造を除くと，記憶セルの概念は，ほかの良く知られているファイル構造の定義では正しく使われていない．すなわち，セルの大きさが一つの場合には任意に固定し(たとえば，索引順序，索引乱ファイル構造では$s=1$)，また別の場合には非常に柔軟に(たとえば，逆ファイル構造では$s=|F|$)している．これらのファイル構造に対し，セルの大きさをもっと合理的に指定できるような定義を提案せよ．順次および並列アクセスアルゴリズムに対して，新しい定義とセルの大きさの指定との影響を示せ．

5.　一般化したファイル構造に対して，ディレクトリとファイルとを構築するプログラムの集合を開発せよ．

6.　セルエントリ，セル識別名，および，ディレクトリ探索の操作子に対するディレクトリのデコーディングプログラムの集合を作成せよ．

7.　順次アクセスアルゴリズムに対するプログラムを作成せよ．そのプログラ

ムへの入力は，キーワードの論理和と一般化したファイル構造とである．そのプログラムからの出力は，各キーワードのリストをたどったときの状態を示すスナップショットの列である．

8. 並列アクセスアルゴリズムを実現せよ．このアルゴリズムへの入力は，キーワードの和標準形(DNF)である．DNF内には少くとも2つの論理積があり，各論理積には3つのキーワードがあるものと仮定せよ．6.5.3節に示した例を使って，このアルゴリズムをテストせよ．アルゴリズムの動きを示すために，同じ節の表6.1およびに6.2示したようにレコードのアクセスに関するスナップショットの列を出さなければならない．最後に，新しい例題のファイルに対して，いくつかの結果を生成せよ．

9. (F.A. Pomeranz) 並列アクセスアルゴリズムにおいて，ファイル探索に対する主キーワードの選択は，ディレクトリ内のレコード数に基づいて決める．論理積の異なったキーワードは異なったセルに対する主キーワードになってもよいけれども，セルに対して(最小のレコード数をもっていたので)論理積の主キーワードとして一度選ばれたキーワードは，そのセルの主キーワードとしてずっと使われている．論理和のキーワードのレコード数を監視することにより，また，そのキーワードを含むレコードが検索される毎にキーワードの対応するレコード数を計算することにより，ファイル探索の途中で主キーワードのレコード数が他のキーワードのレコード数よりも大きくなってしまうことを見出すかもしれない．このような状況が万一生ずると，直ぐに主キーワードを切り替える．すなわち，あるキーワードのレコード数が最小になったならば，いつでもそのキーワードを新しい主キーワードとして動的に指定する．

動的な主キーワードの選択がどのように働くのかを示せ．また，検索の精度とレコード数の計算のオーバーヘッドの改善を示せ．検索の精度の向上は，可動ヘッド機器の読み込み/書き出し機構の再位置づけを犠牲にして行なわれるのか．レコードの番地が単調でないようなファイルにおいても，この改善は得られるのか．

10. (E. Wong および T.C. Chiang) 多重リストおよびセル状多重リストファイルにおいて絶対精度を得ることの難しさの一つは，これらのファイルのリストがレコードを共有していることである．すなわち，これらのリストは共通部分をもつ．一般化したファイル構造をもっと巧みに適用し，キーワードリストの考え方を賢明に解釈すれば，2つのリストが共通のレコードを

もたないような多重リストあるいはセル状多重リストのファイル構造になるであろう．

m 個のキーワードをもつ一般化したファイル構造を考えよう．ブール代数で良く知られているように，2^m 個の最小項が可能である．すなわち，各々の最小項に，m 個の変数が一つづつ現われる．キーワードを変数と見做すと，キーワードの 2^m 個の最小項が可能である．また，ブール代数により，いくつかのキーワードを含んでいる与えられたレコードに対して，レコードは唯一つの最小項だけを満足することがわかる．さらに，すべての最小項が，それを満足するレコードをファイルにもっているわけではない．それにもかかわらず，ファイルに満足すべきレコードをもっているこれらの最小項を**アトム**という．各アトムに対して，それを満足するレコードがファイル内に少くとも一つあることは，明きらかである．さらに，アトムにより，ファイルは相互排除的なレコードのリストに分割される．ディレクトリの中でキーワードの代りにアトムを保持していれば，リストが共通部分をもたない一般化したファイル構造をもてる．新しい一般化したファイル構造では，キーワードリストの概念はアトムリストに拡張される．

この方法を例を使って説明せよ．キーワードの数が減少したり増大したりすることによって生ずるディレクトリのエントリの消去および追加に関する問題を研究せよ．

11.　6.6.1節に概略を示した3つの更新アルゴリズムに対するプログラムを作成せよ．このアルゴリズムの働きを示すために，いくつかのテストの例題を用意せよ．各々のテストでは，ファイルのディレクトリを，更新の前と後で表示しなければならない．

12.　6.6.3.2節で概略を示した屑集めアルゴリズムに対するプログラムを作成せよ．このプログラムへの入力は，ファイルのレコードでそのいくつかは削除タグをもっている．このプログラムの出力は，削除タグをもたないレコードの新しいファイルである．ファイルのディレクトリも，新しいエントリを反映していなければならない．屑集めアルゴリズムが，ある場合にはフェーズⅠだけ，また別の場合にはフェーズⅠとⅡとを必要とすることを示すためには，特別の努力をしなければならない．

13.　並列アクセスアルゴリズムのステップ**5.**で与えられた問い合せに対して検索してきたレコードを調べるのに必要なプログラムは，図6.28に概略を示したアクセス制御のための検証機構として使うことができることを示せ．

第7章

むすび

将来のオペレーティング・システムは，主としてデータベース管理向きに作られるであろう．すなわち，これらのシステムは，主記憶と仮想空間，入力/出力操作，データチャネルとバッファ，機器や端末などの物理的資源を管理しなければならないだけでなく，論理的および物理的資源の構造，保護および使用にも関心を向けなければならなくなる．このようなオペレーティング・システムの見方をすると，システムは前述の章で議論してきた多くのシステムの機能を十分実現できるように総合的かつ精巧になってゆくであろう．このために，**図7.1**にデータベースシステムの構成の概略を示しておく．そのシステムの要素の役割を参照すれば，これらの章に述べた素材とこのシステムとの関連をつけることができよう．

ジョブスーパーバイザ(1)はシステムに対するジョブを受け付け，ユーザにその出力を分配する．オフラインジョブ入出力(第3章3.2.4.1節)と遠隔ジョブ投入(第3章3.3.2.2節)に対して，ジョブスーパーバイザは，1個以上の衛星計算機とのインターフェースになる．オンラインプログラミングが実現されているときには，ジョブスーパーバイザは，各ユーザに対して(ジョブの)入力および出力バッファの対を与えそこでユーザのプログラムの文，データやコマンドを蓄え編集する(第3章3.3.1.1節)ことができるように管理をする．ユーザのジョブはバッファの中に“積み上げられる”ので，ジョブ入力の前処理，ジョブ出力の後処理，および，オンラインコマンドの解釈が可能となる(第3章3.3.1.3.2節)．この処理と解釈のいくらかは，ジョブの投入時に，たとえば，衛星計算機システム(第3章3.3.1.2節)あるいはもっと強力な計算機システムが使える中央集中化した場所(第3章3.2.4.2および3.3.2.1節)でも実行してもよい．大量のジョブを，多重投入で，低速の単位レコード型機器を介して処理するために，ジョブスーパーバイザは，ジョブスプーリングシステム(第3

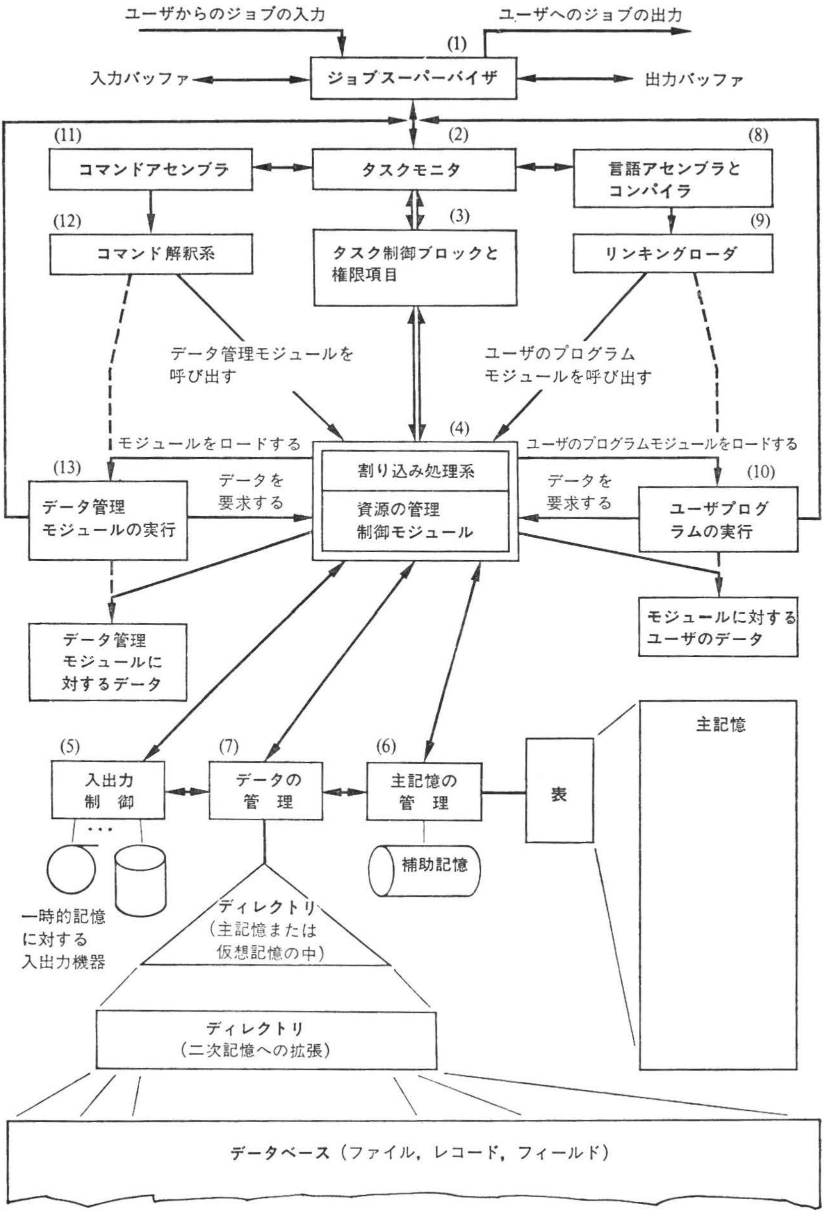


図7.1 データベース・システムの構成

章3.3.2.3.1節)を必要とするかもしれない．ジョブの入力/出力が完了すれば，ジョブは処理のために並べられる．

システムの資源をもっと有効に使い制御するために，総括的なジョブをタスクに分解する．各タスクに対しては，システムはタスク制御ブロックと権限項目(3)とを作る．タスク制御ブロックは，タスクの実行をスケジュールするためにタスクモニタ(2)が使う．よりよい一巡時間(特に応答時間)を保証し，システム資源の平衡のとれた使い方をするために，ある時間刻みに基づいた順番まわしの手続きをタスクモニタが使う (第4章4.2および4.3節；第3章3.2.7節)．

権限項目は，資源の管理制御モジュール(4)が論理的および物理的資源へのアクセスを制御するために使う(第6章6.7節)．タスクは資源に対するアクセスの要求を動的に出してもよいので，これらの要求はSVCの(機械)命令を介して割り込み処理系が受け取る(第5章5.2.2節)．その要求が物理機器，記憶装置あるいはデータセットに対するものであろうと，資源の管理制御モジュールは，要求者のアクセスの属性とシステムが使える資源とに基づいて，それらの要求の承認または拒否を決める．

要求が物理的機器と入出力操作に対するものであれば，入出力制御システム(5)が呼び出される．このシステムでは，機器を検証し，バッファの割り当てまたは切り替えをし，チャネルを並行処理のために使う．(第2章2.4節；第5章5.2.2.2および5.2.2.3節)．

要求が仮想空間と主記憶とに対するものであれば，主記憶管理システム(6)が呼び出される．このシステムでは，補助記憶使用表および物理記憶使用表を参照して，プログラムの再配置と記憶の割り当てを含むページング操作を実行できる(第5章5.2.2および5.2.2.1節)．

要求がデータセットに対するものであれば，データ管理システムが呼び出される(7)．ディレクトリのデコーディングの後，ファイルの探索を行なう．データベースの更新と屑集めは，定期的に行なわれる(第6章6.6.1および6.6.3節)．

データ管理システムは，また，物理的および論理的入出力操作に対するIOCSも利用する．しかしながら，IOCSを使うとき，データ管理システムは補助記憶にアクセスするもっと有効な戦略を採用して，高精度，最小の冗長度，および短いアクセス時間が達成できるようにする(第6章6.5.2節)．

要求した資源が使用可能なときは，機密保護の手続きを行なわなければならない(第6章6.7.3.5節)．データやプログラムを要求者の仮想空間に分配する

ために，主記憶管理システムがインターフェースとなる．この場合，仮想空間に資源の(番地を)写像するために，仮想アクセス法のような特殊な扱いが必要であろう(第5章5.2.2.4節)．

プログラムの準備と実行は，システムの重要な機能である．アセンブラ(8)，リンキングローダ(9)，および，プログラム実行フェーズ(10)の関係は，(会話型またはオンラインの)コマンド言語アセンブラ(11)，解釈系(12)および，コマンドモジュール実行フェーズ(13)の関係と非常に良く似ている．この相似性は，資源への要求を出すとき，タスクモニタ，割り込み処理系，資源の管理制御モジュールと相互作用をするときの方法に良くあらわれている．

アセンブリ・システム(第1章)およびコマンド処理系(第3章3.3.1.3.2節；第6章6.7.3.4節)は，サブルーチンのリンキングとローディングのために静的あるいは動的なリンキングローダのどちらかを利用する(第1章1.5節；第5章5.2.2.6節)．さらに，プログラムの実行中は，システムマクロの形式をとるプログラムの要求と呼び出しが行なわれるかもしれない(第5章5.2.2.3節；第6章6.7.3.4節)．

前出の章は一意的なシステムの概念と機能の上に構成されているけれども，これらの概念と機能とは大規模かつ高度なシステムの中でまとめてかつ総合的に出現しうるし，またするであろう．

索　　引

あ　行

I-C 重ね合わせ……77
IBM 360/67……218,232
——360 BOS……132
——360 MFT……185
——360 MVT……187
——360 TOS/DOS……178
——370……218
アクセス制御行列……364
アセンブラ……1
アセンブリ言語……3
アセンブリ・システム……2
アタッチマクロ……187
安　全……201,217
一次元仮想記憶システム……231
一巡時間……154,176
一様可変型（UV 型）……308
一様固定型（UF 型）……306
一様出現……300
一般化したファイル構造……314
一般化したレコード構成……299
意味処理……21
入口初期設定……207
入力前処理……150
インターバルタイマ……249
衛星計算機のオペレーティング・システム……146
遠隔ジョブ投入……154
OS/VS 2……218
OS/360-ASP……141
応答時間……176
オーバーレイ……44
置き換えアルゴリズム……246
オフライン一括処理オペレーティング・システム……111
オフライン・オペレーティング・システムの階層的構成……113
オフライン・ジョブ入出力……140
親システム……132
オンライン一括処理システム……142
オンラインジョブ入出力……139
オンラインプログラミング……107,142

か　行

開始サブタスク……188
外部記号ディクショナリ……12
外部定義記号（REFS）……276
書き出しサブタスク……188
核……114,117
仮想アクセス法……269
仮想記憶アクセス法（VSAM）……239
仮想記憶システム……224
仮想空間……43,223
仮想空間表……231
仮想番地……43,223
間接結合システム……153
キーワード……259
記憶折り重ね……173
記憶変換……177,223
記憶保護……359
記憶割り当て機構……219
機器オーダー……73
機器制御ブロック……93
危険部分……212
記号機械命令……3,5
記号参照表……44
記号装置……94
記号装置名表……96
記号定義（DEFS）……277
記号表……5,18
擬似命令……4,9
擬似命令ディクショナリ……24
機密保護の手続き……370
逆ファイル……316
共通資源へのアクセスの制御……190
近隣探索……163
屑集め系……352
屑集めのアルゴリズム……353
計算限界……65

*K*リスト ……295
権限項目 ……364
言語処理系サブシステム ……125
言語ライブラリ更新サブシステム ……123
言語ライブラリ内容表 ……124
原始プログラム ……2
更新アルゴリズム ……349
構造的な移動 ……352
構文解析 ……17,19
コマンド言語 ……119
コントロールセクション ……9
コンパイラ ……129

さ　行

サイクルスティーリング ……71
再構成 ……308
再入可能型 ……195,206,274
再配置 ……43,221
索　引 ……295
索引順序ファイル ……314
索引乱ファイル ……314
削除タグ ……350
サブシステムモニタ ……116
サブタスク ……187
サブルーチンライブラリ ……123
CDC 3500 ……218,227
GIS ……294
時間刻み ……174
字句解析 ……17
システムカタログ (Syscat) ……260
システム生成 ……116
システム操作表 ……145
システムデッドロック ……199
　——の検出と回復 ……203
システム掃き出しと再開始 ……116
システムライブラリ更新サブシステム ……118,122
システムライブラリ装置 ……118
システムライブラリ内容表 ……118
システムレコード ……304
システムローダ ……116,133
実行可能状態 ……148
修飾ファイル名 ……260
主キーワード ……336
主システムタスク ……187
主シリンダマップ ……159
主タスク ……187
従属対 ……299
出力後処理 ……150
順次アクセスアルゴリズム ……332
純手続き ……206
使用状態 ……149
ジョブシリンダマップ ……159
ジョブ情報ブロック (JIB) ……179
ジョブスケジューラ ……151
ジョブステップ ……187
ジョブスプーリングシステム ……156,166
ジョブ制御 ……134
ジョブ制御言語 ……119
ジョブ制御表 ……160
ジョブの自動化 ……108
ジョブ待ち行列要素 ……161
所有権 ……365
所有者 ……365
上位対 ……299
常駐型制御プログラム ……114
処理量 ……76,84
シリンダ探索アルゴリズム ……161
指令的出現 ……300
スーパーバイザ ……116,120
スワッピング ……186
静的再配置 ……221
静的リンキングローダ ……221
精度 (探索の) ……334
セグメント ……232
　——番号 ……232
　——表 ……232
セマフォ ……197
セ　ル ……313
　——の大きさ ……313
セルエントリ操作子 ……329
セル識別名操作子 ……330
セル状多重リストファイル ……321
セル番地 ……313
線型仮想記憶システム ……231
線型セグメント化仮想記憶システム ……233
全域探索 ……163
全域的レコード構造仕様 ……304
選択的可変型 (OV 型) ……307
選択的固定型 (OF 型) ……307

選択的出現 ……300
送信マクロ ……189
相対番地……38
装置制御ブロック……95

た　行

多重タスキング ……284
多重パスアセンブラ……17
多重プログラミング ……176
多重プログラミング・オペレーティング・システムの階層構造 ……183
多重リスト ……319
多重リストファイル ……318
タスク ……187,284
——の同期 ……188
タスク開始部 ……248,256
タスク制御ブロック（TCB） ……244
タスクディクショナリ（TDY） ……278
単層記憶概念 ……237
断片化（プログラムの）……219
端末入出力終了割り込み応答 ……257
端末待ち（SVC） ……256
遅延形の擬似命令……15
遅延の記号……15
チェイニング……43
チャネルコマンド……73
チャネルコマンド語……71
チャネルスケジューラ ……134
チャネルプログラム ……252
チャネル待ち行列 ……178
中央計算機のオペレーティング・システム ……147
直上上位対 ……299
直接結合システム ……141
直接探索 ……163
直下従属対 ……299
2パスアセンブラ ……6
詰め操作（プログラムの）……220
釣り合いのとれたプログラム……61
TSS ……218
TSOS ……218
TDMS ……294
DATRIX ……294
ディスクリプタ・ベース・レジスタ（DBR）……240
ディレクトリ ……259,314
ディレクトリ探索操作子 ……330
データの機密保護 ……358
データチャネル……72
データベースのレコード ……295
データ保全性 ……311
テープ誤りブロック（TEB） ……179
DEF 表 ……278
出口戻り ……207
手続き保護 ……359
デッドロック ……191
デブロッキング……58,100
問い合わせ ……327
導出型紙 ……311
動的結合ローダ ……274
動的ローディング……44

な　行

内部再配置用記号ディクショナリ……39
内部テキスト表……19
内容表（TOC） ……269
二次元仮想記憶システム ……232
二次記憶セル ……313
入出力干渉……72
入出力限界……64
入出力検証サブシステム（IOVS） ……99
入出力制御システム（IOCS）…75,86,101
入出力操作解釈系……97
入出力操作実行系……97
入出力デッドロック ……181
入出力バッファリング・サブシステム（IOBS）……100,122
入出力待ち（SVC） ……256
入出力要求待ち行列 ……178
入力前処理 ……150

は　行

HASP ……158
Honeywell 645 ……218
パスの集合 ……250
バックグラウンドプログラム領域 ……182
ハッシュ化アルゴリズム……50
バッファ……61
——の切り換え……56
——の割り当て……67

出力——……56
入力——……56
バッファプール……56
非常駐型制御プログラム……118
PI/O マクロ……252
p 操作……197
ファイル……258,297
——のファイル（FOF）……261
——の保全性……181
ファイル共用システム……153
ファイルシステム……269
ファイル制御ブロック（FCB）…106,263
ファイル探索関数……330
ファイル名……297
VS 370-JES 2……141
v 操作……197
フォアグラウンドプログラム領域……
第1——……182
第2——……182
副タスク……284
物理 IOCS……102
物理記憶……223
物理記憶空間表……231
物理記憶使用表……231
物理記憶表……231
物理記憶利用表……226
物理装置ブロック（PUB）……179
物理的な保護とアクセス制御機構……359
物理入出力 SVC……252
物理入出力応答ルーチン……252
物理入出力制御サブシステム（PIOCS）……257
物理番地……223
物理レコード……55,304
物理レコード構成……304
プログラム
——の断片化……219
——の詰め操作……220
プログラムのオーバーレイ……227
プログラムフェーズ実行サブシステム 130
プログラムロード……281
ブロッキング……58,100
ブロッキング係数……59
ブロック……58,224
ブロック数……226
平均アクセス時間……66
並行処理……67
並列アクセスアルゴリズム……335
ページ……224
ページ数……226
ページ表……232
ページフォールト……229,246
ページフォールト割り込み応答ルーチン……246
ページローダ……282,283
ページロード……281
ページング……231
ページングアウト操作……231
ページングイン操作……231
ページング処理……229
ページングチャネル終了割り込み応答ルーチン……246
変換関数……223
変換表……223
ポインタ……295
補助記憶使用表……231

ま　行

待ち行列挿入……191
待ち行列除去……191
待ち状態……149
待ちマクロ……189
まばらに使用される線形セグメント化仮想記憶システム……237
MULTICS……218
命令ディクショナリ……22
目的プログラム……2,38
モニタ……116,122

や　行

UNIVAC Spectra 70/46……218
ユーザカタログ（Usercat）……266
呼び出し系列……207
読み込みサブタスク……187

ら　行

リスト……295
——の順次処理……332
——の並列処理……338
リンキングローダ……42,125

リンケージエディタ ……135
レコード ……295
レコード型紙 ……304
レコード構成 ……297
レコード制御情報 ……304
レコード制御ブロック ……304
REF表 ……277
レベル番号 ……299
ロケーションカウンタ ……9
ロケーションカウンタ・ディクショナリ ……26
ロード時再配置可能多重プログラミング・システム ……177
ロードファイル ……126
ロードモジュール ……42
論理 IOCS ……102
論理装置ブロック (LUB) ……179
論理的アクセス制御モデル ……362
論理的な保護とアクセス制御機構 ……359
論理入出力制御サブシステム (LIOCS) ……259
論理レコード ……58,304
論理レコード構成 ……304

わ　行

割り込み ……98
割り込み応答ルーチン ……245
割り込み解析部 ……245
割り込み処理系 ……133,148
割り出し ……115

《訳者略歴》

斎藤信男
さいとうのぶお

昭和15年　松本市に生まれる
昭和39年　東京大学工学部計数工学科卒業
昭和41年　東京大学大学院工学系研究科修士課程修了
昭和41年　通産省工業技術院電気試験所（現電子技術総合研究所）入所.
昭和45年～昭和46年　スタンフォード大学に留学(科学技術庁在外研究員).
昭和49年　筑波大学電子・情報工学系専任講師
昭和50年　筑波大学電子・情報工学系助教授
昭和53年　慶応義塾大学工学部数理工学科助教授
現　在：　慶応義塾大学工学部数理工学科助教授・工学博士

システム・プログラミング

昭和55年4月20日　初版発行　　©1980　科学技術出版社

検　印	廃　止

著　者　David K. Hsiao
訳　者　斎　藤　信　男
発行者　村　越　勝　弘
編　者　日本コンピュータ協会
東京都文京区本郷3丁目5—5
発行所　**科学技術出版社**
東京都文京区本郷3丁目5—5
〒113　電話 (03) 813—6920番

乱丁本・落丁本はお買い求め書店にてお取りかえいたします.　　Printed in Japan